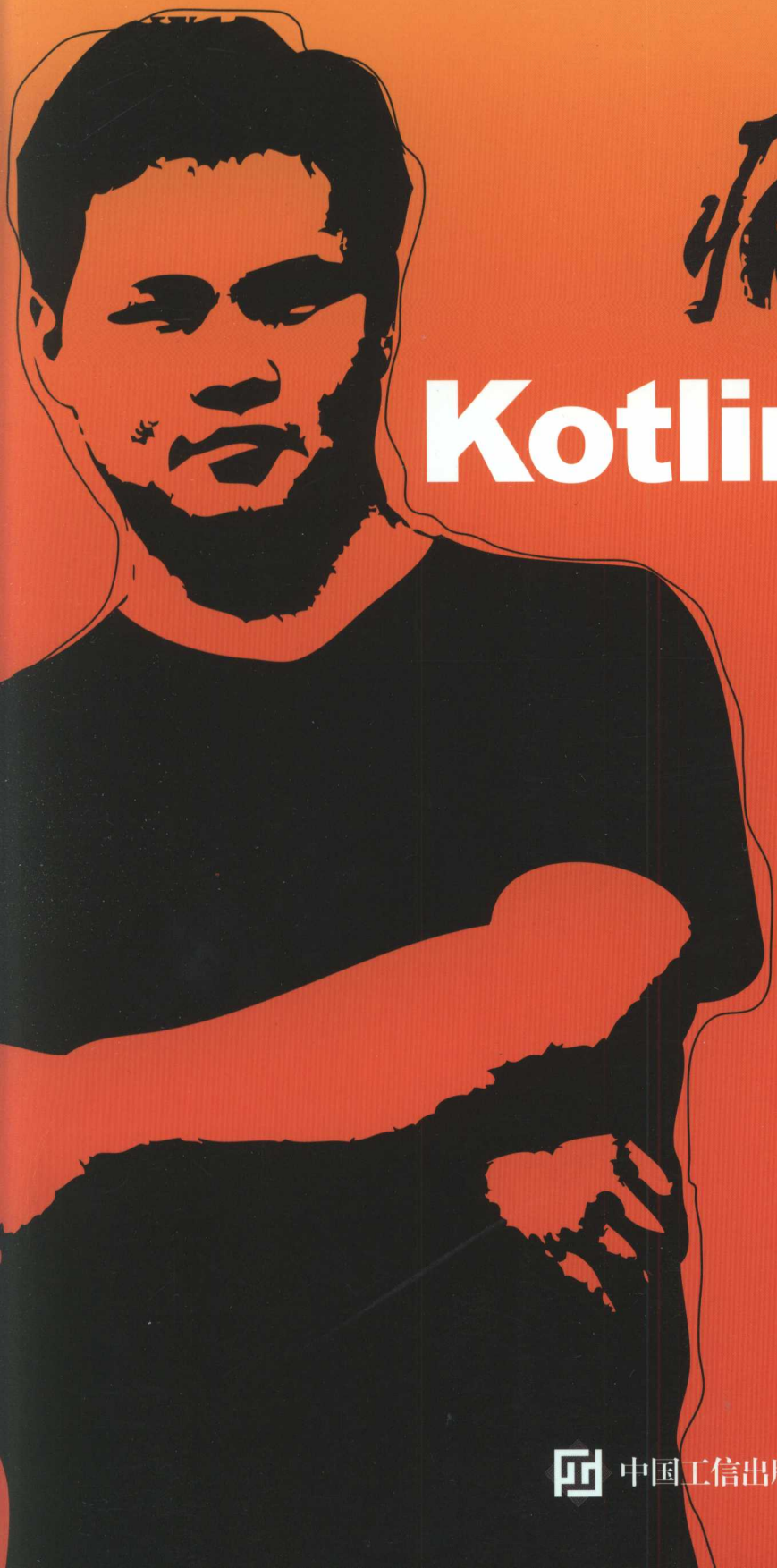
版权注意事项：
1、书籍版权归作者和出版社所有
2、本PDF仅限用于个人获取知识，进行私底下的知识交流
3、PDF获得者不得在互联网上以任何目的进行传播
4、如觉得书籍内容很赞，请购买正版实体书，支持作者
5、请于下载PDF后24小时内删除本PDF。

Broadview
www.broadview.com.cn

FK 疯狂软件教育 www.fkjava.org 标准教材



疯狂Kotlin讲义

李刚 编著

疯狂源自梦想
技术成就辉煌

疯狂源自梦想
技术成就辉煌

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

全国行业大奖得主
电子工业出版社优秀作者



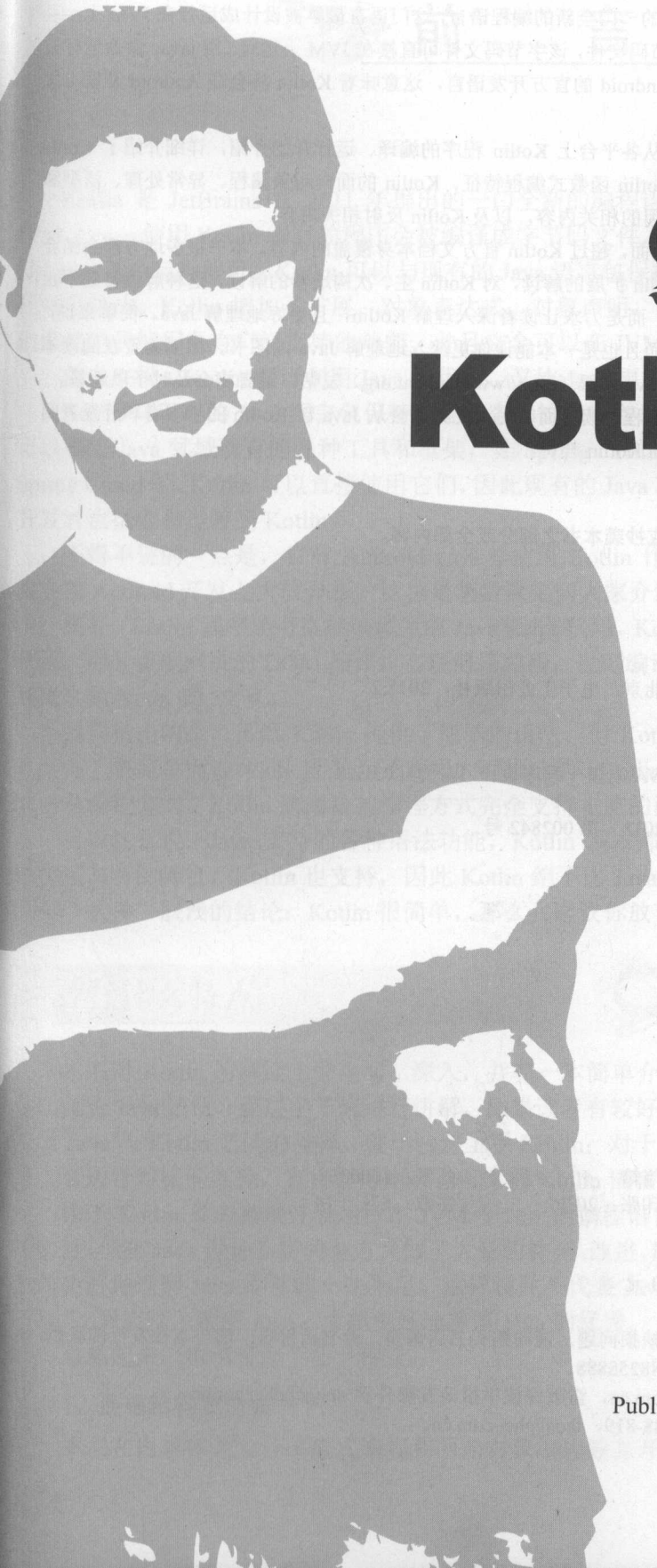
作者简介

李刚，十余年软件开发从业经验，疯狂软件教育中心教学总监。疯狂Java实训营创始人，疯狂Java体系原创图书作者。广东技术师范学院计算机科学系兼职副教授，CSDN特邀讲师。培训的学生已在腾讯、阿里、华为、IBM、网易、唯品会、电信盈科等名企就职。国内知名高端IT技术图书作家，已出版《疯狂Java讲义》《疯狂Android讲义》《轻量级Java EE企业应用实战》《疯狂前端开发讲义》《疯狂HTML 5/CSS 3/JavaScript讲义》《疯狂iOS讲义（基础篇）（提高篇）》《疯狂XML讲义》《经典Java EE企业应用实战》《Struts 2.x权威指南》等著作。其中疯狂Java体系图书均已沉淀多年，赢得极高的市场认同，多次重印，多部著作印刷数量超过10万册，并被多所“985”“211”院校选作教材，部分图书已被翻译成繁体中文版，授权到中国台湾地区。

新浪微博：
weibo.com/lgjava　@疯狂软件李刚

扫码关注“疯狂图书”微信号可获得：

· 本书配套代码下载
· 作者不定期答疑服务



疯狂 Kotlin讲义

李刚 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Kotlin 是 JetBrains 在 2011 年推出的一门全新的编程语言，这门语言最早被设计成运行在 JVM 上——使用 Kotlin 编写的程序会被编译成字节码文件，该字节码文件可直接在 JVM 上运行（用 java，命令运行）。目前 Google 已推荐使用 Kotlin 作为 Android 的官方开发语言，这意味着 Kotlin 将会在 Android 开发上大放异彩。

本书全面介绍了 Kotlin 的语法。从各平台上 Kotlin 程序的编译、运行开始介绍，详细介绍了 Kotlin 的基本语法，Kotlin 的数组和集合，Kotlin 函数式编程特征，Kotlin 的面向对象编程、异常处理、泛型和注解，还介绍了 Kotlin 与 Java 混合调用的相关内容，以及 Kotlin 反射相关内容。

本书对 Kotlin 的解读十分系统、全面，超过 Kotlin 官方文档本身覆盖的内容。本书很多地方都会结合 Java 字节码进行深入解读，比如对 Kotlin 扩展的解读，对 Kotlin 主、次构造器的解读，这种解读目的不止于教会读者简单地掌握 Kotlin 的用法，而是力求让读者深入理解 Kotlin，且更好地理解 Java。简单来说，本书不仅是一本 Kotlin 的学习图书，而且也是一本能让你更深入地理解 Java 的图书。如果读者在阅读本书时遇到了技术问题，可以登录疯狂 Java 联盟（http://www.crazyit.org）发帖，笔者将会及时予以解答。

本书为所有打算深入掌握 Kotlin 编程的读者而编写，尤其适合从 Java 转 Kotlin 的学习者和开发者阅读，也适合作为大学教育、培训机构的 Kotlin 教材。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

疯狂 Kotlin 讲义 / 李刚编著. —北京：电子工业出版社，2018.2
ISBN 978-7-121-33459-7

Ⅰ. ①疯… Ⅱ. ①李… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312.8

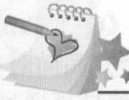
中国版本图书馆 CIP 数据核字（2018）第 002842 号

策划编辑：张月萍
责任编辑：葛　娜
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：20.25　　字数：571 千字
版　　次：2018 年 2 月第 1 版
印　　次：2018 年 2 月第 1 次印刷
印　　数：3500 册　　定价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。



前　言

Kotlin 是 JetBrains 在 2011 年推出的一门全新的编程语言，这门语言最早被设计成运行在 JVM 上——使用 Kotlin 编写的程序会被编译成字节码文件，该字节码文件可直接在 JVM 上运行（用 java 命令运行即可）。Kotlin 可以与现有的 Java 语言包保持完全兼容，而且 Kotlin 代码比 Java 代码更简洁。Kotlin 增加了扩展、对象表达式、对象声明、委托等 Java 原本不支持的功能，它们都是现代编程语言广泛支持的功能，并且完全可以在 JVM 上运行。

简单来说，Kotlin 既可利用 Java 的优势，又比 Java 更简洁。

Kotlin 与现有的 Java 语言包保持完全兼容，这意味着 Kotlin 不是一门简单的语言，它完全可以利用 Java 领域现有的各种工具和框架，如 Spring、Hibernate、MyBatis、Lucene、Hadoop、Spring Cloud 等。Kotlin 可以直接使用它们，因此现有的 Java 项目完全可以采用 Kotlin 开发，Java 开发者也很容易过渡到 Kotlin。

不得不说的一点是，目前 Android 已推荐使用 Kotlin 作为官方开发语言，这意味着 Kotlin 将会在 Android 开发上大放异彩，这也是笔者决定向大家介绍这门语言的重要原因之一。

此外，Kotlin 程序还可直接编译生成 JavaScript 代码，Kotlin 程序既可编译成前端 JavaScript 代码，用于实现网页的 DOM 操作，实现前端编程；也可编译成后端 JavaScript 代码，与服务端技术（如 Node.js）交互。

需要指出的是，虽然 Kotlin 提供了简洁的语法，但 Kotlin 的功能并不简单，Kotlin 从来就不是为了更简单而设计的，而是为了更强大而设计的。Kotlin 既支持函数式编程方式，也支持面向对象编程方式。Kotlin 的函数式编程方式完全支持主流的函数和闭包，语法功能非常丰富。

可以这么说：Java 支持的各种语法功能，Kotlin 基本都支持；Java 不支持的很多现代编程语言所具有的特征，Kotlin 也支持，因此 Kotlin 绝不比 Java 更简单。如果读者相信网络上某些所谓“大神”肤浅的结论：Kotlin 很简单，那么我建议你放弃阅读这本书。

本书有什么特点



本书对 Kotlin 的解读十分全面、深入，并非一本简单介绍 Kotlin 语法的图书，在很多地方都会结合 Java 语法、底层字节码进行讲解。如果读者有较好的 Java 功底，阅读本书能更清晰地看清 Java 与 Kotlin 之间的差异，便于快速上手 Kotlin；对于没有 Java 功底的读者，可选择忽略将二者进行对比的部分，直接学习本书也可掌握 Kotlin 语言的编程。

由于 Kotlin 最先被设计成运行在 JVM 平台上的编程语言，因此 Kotlin 具有和 Java 天然的相似性，但在 Java 设计不足的地方又做了大量的补充、改进，所以本书也能让你更好地理解 Java，以及更好地理解 Java 存在的一些不足。这样说并不代表 Java 不优秀，“知其雄，守其雌，为天下谿”，只有深入理解 Java，才能更好地感悟 Java 的优秀。

总结起来，本书有如下几个特点。

1. 逻辑结构更合理

本书在内容体系上将函数式编程和面向对象编程独立开来，先介绍函数式编程部分，再介

绍面向对象编程，更符合 Kotlin 语言本身的知识体系。而不像某些资料一会儿函数，一会儿面向对象，搅得读者晕头转向。实际上，无论是经典的图书如《C++ Primer》，还是 Swift 官方文档（甚至 Kotlin 官方文档），几乎都没见过一会儿函数、一会儿面向对象这样介绍的。

在介绍知识时，本书会先详细讲解各种知识点的理论，然后再通过示例演示 Kotlin 各理论的用法，将知识点融合在示例中，符合读者的认知、学习规律。

另外，整本书的知识具有和《疯狂 Java 讲义》大致相同的脉络，所以《疯狂 Java 讲义》的读者会很容易上手。

2. 讲解深入本质

Kotlin 是 JVM 语言，所以其很多东西其实是受到 Java 的影响的。书中会对一些看似奇怪的语法从字节码文件层次进行剖析，让读者更好地理解 Kotlin 与 Java 的对应关系。

比如，主构造器和次构造器到底是什么？Java 构造器并不区分主次，为什么 Kotlin 搞出这两个东西？主、次构造器为何要委托父类构造器？委托父类构造器时为什么存在区别？主、次构造器生成字节码之后到底对应 Java 的哪个部分？这些知识在本书 7.5 节有深入讲解。

再比如，Java 本身不支持扩展，那么 Kotlin 的扩展是如何在 JVM 上运行的？难道 Kotlin 改造了 JVM 吗？Java 本身不支持扩展，那么 Java 是否可以调用 Kotlin 扩展的成员？这些问题需要从字节码层次进行剖析，本书在 8.1 节有深入讲解。

还有，Java 泛型的上限、下限的本质是什么？Kotlin 泛型的声明处型变和使用处型变的本质是什么？与 Java 的对应关系是怎样的？Java 本身并不支持声明处型变，那为何 JVM 能支持 Kotlin 的声明处型变？这些问题可以在本书第 10 章中找到答案。

3. 知识内容更全面

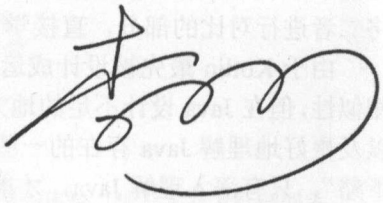
本书内容超过 Kotlin 官方文档本身所覆盖的知识，比如介绍反射的章节就超过了 Kotlin 官方文档内容。本书反射部分不仅更详细地介绍如何获得类、函数、属性的引用，而且真正从 API 级别介绍 KClass、KCallable、KFunction、KProperty，KProperty0、KProperty1、KProperty2 的用法，以及它们的内在关联，并实实在在地教读者掌握如何用 Kotlin 反射动态创建对象、动态调用方法。

本书写给谁看



本书为所有打算深入掌握 Kotlin 编程的读者而编写，尤其适合从 Java 转 Kotlin 的学习者和开发者阅读，也适合作为大学教育、培训机构的 Kotlin 教材。

本书程序文件请从 www.broadview.com.cn/33459 下载。



2017-12-3

目 录 CONTENTS

第 1 章 Kotlin 语言与开发环境 1

1.1 Kotlin 语言简介 2
1.1.1 服务端的 Kotlin 2
1.1.2 使用 Kotlin 开发 Android 应用 2
1.1.3 Kotlin 用于 JavaScript 3
1.2 使用命令行编译、运行 Kotlin 3
1.2.1 下载和安装 Kotlin 的 SDK 3
1.2.2 第一个 Kotlin 程序 4
1.2.3 编译、运行 Kotlin 程序 5
1.3 使用 IntelliJ IDEA 编译、运行 Kotlin 6
1.4 使用 Eclipse 编译、运行 Kotlin 8
1.5 本章小结 10

第 2 章 Kotlin 的基础类型 11

2.1 注释 12
2.1.1 单行注释和多行注释 12
2.1.2 文档注释 12
2.2 变量 14
2.2.1 分隔符 15
2.2.2 标识符规则 16
2.2.3 Kotlin 的关键字 17
2.2.4 声明变量 19
2.3 整型 21
2.4 浮点型 23
2.5 字符型 24
2.6 数值型之间的类型转换 25
2.6.1 整型之间的转换 25
2.6.2 浮点型与整型之间的转换 28
2.6.3 表达式类型的自动提升 28
2.7 Boolean 类型 30
2.8 null 安全 31
2.8.1 非空类型和可空类型 31
2.8.2 先判断后使用 32
2.8.3 安全调用 32
2.8.4 Elvis 运算 33
2.8.5 强制调用 34
2.9 字符串 34
2.9.1 字符串类型 34
2.9.2 字符串模板 35
2.9.3 Kotlin 字符串的方法 36
2.10 类型别名 37
2.11 本章小结 38

第 3 章 运算符和表达式 39

3.1 与 Java 相同的运算符 40
3.1.1 单目前缀运算符 40
3.1.2 自加和自减运算符 41
3.1.3 双目算术运算符 41
3.1.4 in 和!in 运算符 42
3.1.5 索引访问运算符 43
3.1.6 调用运算符 43
3.1.7 广义赋值运算符 44
3.1.8 相等与不等运算符 44
3.1.9 比较运算符 45
3.2 位运算符 46
3.3 区间运算符 48
3.3.1 闭区间运算符 48
3.3.2 半开区间运算符 49
3.3.3 反向区间 49
3.3.4 区间步长 49
3.4 运算符重载 50
3.4.1 重载单目前缀运算符 50
3.4.2 重载自加和自减运算符 51
3.4.3 重载双目算术运算符 51
3.5 本章小结 52

第 4 章 流程控制 53

4.1 顺序结构 54
4.2 分支结构 54
4.2.1 if 分支 54
4.2.2 if 表达式 58
4.2.3 when 分支语句 58
4.2.4 when 表达式 61
4.2.5 when 分支处理范围 62
4.2.6 when 分支处理类型 62
4.2.7 when 条件分支 63
4.3 循环结构 63
4.3.1 while 循环 64
4.3.2 do while 循环 65
4.3.3 for-in 循环 66
4.3.4 嵌套循环 66
4.4 控制循环结构 68
4.4.1 使用 break 结束循环 68

4.4.2 使用 continue 忽略本次循环的剩下语句 69
4.4.3 使用 return 结束方法 70
4.5 本章小结 71

第 5 章 数组和集合 72

5.1 数组 73
5.1.1 创建数组 73
5.1.2 使用数组 75
5.1.3 使用 for-in 循环遍历数组 76
5.1.4 使用数组索引 76
5.1.5 数组的常用方法 77
5.1.6 多维数组 80
5.1.7 数组的应用举例 82
5.2 Kotlin 集合概述 85
5.3 Set 集合 88
5.3.1 声明和创建 Set 集合 88
5.3.2 使用 Set 的方法 90
5.3.3 遍历 Set 91
5.3.4 可变的 Set 92
5.4 List 集合 93
5.4.1 声明和创建 List 集合 93
5.4.2 使用 List 的方法 94
5.4.3 可变的 List 95
5.5 Map 集合 95
5.5.1 声明和创建 Map 集合 95
5.5.2 使用 Map 的方法 97
5.5.3 遍历 Map 98
5.5.4 可变的 Map 98
5.6 本章小结 99

第 6 章 函数和 Lambda 表达式 100

6.1 函数入门 101
6.1.1 定义和调用函数 101
6.1.2 函数返回值和 Unit 102
6.1.3 递归函数 103
6.1.4 单表达式函数 104
6.2 函数的形参 105
6.2.1 命名参数 105
6.2.2 形参默认值 106
6.2.3 尾递归函数 108
6.2.4 个数可变的形参 109
6.3 函数重载 110
6.4 局部函数 111
6.5 高阶函数 112
6.5.1 使用函数类型 112
6.5.2 使用函数类型作为形参类型 113
6.5.3 使用函数类型作为返回值类型 114
6.6 局部函数与 Lambda 表达式 115
6.6.1 回顾局部函数 116
6.6.2 使用 Lambda 表达式代替局部函数 116
6.6.3 Lambda 表达式的脱离 117
6.7 Lambda 表达式 117
6.7.1 调用 Lambda 表达式 118
6.7.2 利用上下文推断类型 118
6.7.3 省略形参名 119
6.7.4 调用 Lambda 表达式的约定 120
6.7.5 个数可变的参数和 Lambda 参数 120
6.8 匿名函数 121
6.8.1 匿名函数的用法 121
6.8.2 匿名函数和 Lambda 表达式的 return 122
6.9 捕获上下文中的变量和常量 123
6.10 内联函数 125
6.10.1 内联函数的使用 125
6.10.2 部分禁止内联 126
6.10.3 非局部返回 127
6.11 本章小结 128

第 7 章 面向对象（上） 129

7.1 类和对象 130
7.1.1 定义类 130
7.1.2 对象的产生和使用 132
7.1.3 对象的 this 引用 133
7.2 方法详解 136
7.2.1 方法与函数的关系 136
7.2.2 中缀表示法 137
7.2.3 componentN 方法与解构 138
7.2.4 数据类和返回多个值的函数 140
7.2.5 在 Lambda 表达式中解构 141
7.3 属性和字段 142
7.3.1 读写属性和只读属性 142
7.3.2 自定义 getter 和 setter 144
7.3.3 幕后字段 147
7.3.4 幕后属性 148
7.3.5 延迟初始化属性 149
7.3.6 内联属性 150
7.4 隐藏和封装 151
7.4.1 包和导包 151
7.4.2 Kotlin 的默认导入 153
7.4.3 使用访问控制符 153
7.5 深入构造器 155

7.5.1 主构造器和初始化块 156
7.5.2 次构造器和构造器重载 158
7.5.3 主构造器声明属性 161
7.6 类的继承 161
7.6.1 继承的语法 161
7.6.2 重写父类的方法 164
7.6.3 重写父类的属性 166
7.6.4 super 限定 167
7.6.5 强制重写 168
7.7 多态 169
7.7.1 多态性 169
7.7.2 使用 is 检查类型 170
7.7.3 使用 as 运算符转型 172
7.8 本章小结 174

第 8 章 面向对象（下） 175

8.1 扩展 176
8.1.1 扩展方法 176
8.1.2 扩展的实现机制 179
8.1.3 为可空类型扩展方法 182
8.1.4 扩展属性 182
8.1.5 以成员方式定义扩展 183
8.1.6 带接收者的匿名函数 184
8.1.7 何时使用扩展 186
8.2 final 和 open 修饰符 187
8.2.1 可执行“宏替换”的常量 187
8.2.2 final 属性 188
8.2.3 final 方法 189
8.2.4 final 类 190
8.2.5 不可变类 190
8.3 抽象类 192
8.3.1 抽象成员和抽象类 192
8.3.2 抽象类的作用 195
8.3.3 密封类 196
8.4 接口 198
8.4.1 接口的定义 198
8.4.2 接口的继承 199
8.4.3 使用接口 200
8.4.4 接口和抽象类 202
8.5 嵌套类和内部类 202
8.5.1 内部类 204
8.5.2 嵌套类 207
8.5.3 在外部类以外使用内部类 209
8.5.4 在外部类以外使用嵌套类 209
8.5.5 局部嵌套类 210
8.5.6 匿名内部类 211
8.6 对象表达式和对象声明 212
8.6.1 对象表达式 212
8.6.2 对象声明和单例模式 215
8.6.3 伴生对象和静态成员 217
8.6.4 伴生对象的扩展 218
8.7 枚举类 219
8.7.1 枚举类入门 219
8.7.2 枚举类的属性、方法和构造器 221
8.7.3 实现接口的枚举类 222
8.7.4 包含抽象方法的抽象枚举类 222
8.8 类委托和属性委托 223
8.8.1 类委托 224
8.8.2 属性委托 225
8.8.3 延迟属性 227
8.8.4 属性监听 228
8.8.5 使用 Map 存储属性值 230
8.8.6 局部属性委托 231
8.8.7 委托工厂 233
8.9 本章小结 234

第 9 章 异常处理 236

9.1 异常处理机制 237
9.1.1 使用 try...catch 捕获异常 237
9.1.2 异常类的继承体系 240
9.1.3 访问异常信息 242
9.1.4 异常处理嵌套 243
9.1.5 try 语句是表达式 243
9.2 使用 throw 抛出异常 243
9.2.1 抛出异常 243
9.2.2 自定义异常类 244
9.2.3 catch 和 throw 同时使用 245
9.2.4 异常链 246
9.2.5 throw 语句是表达式 247
9.3 异常的跟踪栈 248
9.4 本章小结 250

第 10 章 泛型 251

10.1 泛型入门 252
10.1.1 定义泛型接口、类 252
10.1.2 从泛型类派生子类 253
10.2 型变 254
10.2.1 泛型型变的需要 254
10.2.2 声明处型变 256
10.2.3 使用处型变：类型投影 258
10.2.4 星号投影 260
10.3 泛型函数 261

10.3.1 泛型函数的使用 261
10.3.2 具体化类型参数 262
10.4 设定类型形参的上限 263
10.5 本章小结 264

第 11 章 注解 266

11.1 Kotlin 注解入门 267
11.1.1 定义注解 267
11.1.2 注解的属性和构造器 268
11.2 元注解 270
11.2.1 使用@Retention 270
11.2.2 使用@Target 271
11.2.3 使用@MustBeDocumented 272
11.2.4 使用@Repeatable 标记可重复注解 273
11.3 使用注解 273
11.3.1 提取注解信息 273
11.3.2 使用注解的示例 274
11.4 Java 注解与 Kotlin 的兼容性 279
11.4.1 指定注解的作用目标 279
11.4.2 使用 Java 注解 281
11.5 本章小结 282

第 12 章 Kotlin 与 Java 互相调用 283

12.1 Kotlin 调用 Java 284
12.1.1 属性 284
12.1.2 void 和调用名为关键字的成员 285
12.1.3 Kotlin 的已映射类型 286
12.1.4 Kotlin 对 Java 泛型的转换 287
12.1.5 对 Java 数组的处理 287
12.1.6 调用参数个数可变的方法 288
12.1.7 checked 异常 289
12.1.8 Object 的处理 289
12.1.9 访问静态成员 290
12.1.10 SAM 转换 290
12.1.11 在 Kotlin 中使用 JNI 291
12.2 Java 调用 Kotlin 291
12.2.1 属性 291
12.2.2 包级函数 292
12.2.3 实例变量 294
12.2.4 类变量 294
12.2.5 类方法 296
12.2.6 访问控制符的对应关系 297
12.2.7 获取 KClass 298
12.2.8 使用@JvmName 解决签名冲突 298
12.2.9 生成重载 299
12.2.10 checked 异常 300
12.2.11 泛型的型变 300
12.3 Kotlin 反射 302
12.3.1 类引用 302
12.3.2 从 KClass 获取类信息 303
12.3.3 创建对象 306
12.3.4 构造器引用 306
12.3.5 调用方法 307
12.3.6 函数引用 308
12.3.7 访问属性值 309
12.3.8 属性引用 311
12.3.9 绑定的方法与属性引用 313
12.4 本章小结 313

CHAPTER

1

第 1 章
Kotlin 语言与开发环境

本章要点

- Kotlin 语言简介
- 服务端 Kotlin 与客户端 Kotlin
- 下载和安装 Kotlin 的 SDK
- 编写第一个 Kotlin 程序
- 编译、运行 Kotlin 程序
- 使用 IntelliJ IDEA 编译、运行 Kotlin
- 使用 Eclipse 编译、运行 Kotlin

Kotlin 是 JetBrains 在 2011 年推出的一门全新的编程语言，这门语言最早被设计成运行在 JVM（Java 虚拟机）上——使用 Kotlin 编写的程序会被编译成字节码文件，该字节码文件可直接在 JVM 上运行(用 java 命令运行)。Kotlin 可以与现有的 Java 语言包保持 100%的兼容性，而且 Kotlin 代码比 Java 代码更简洁、更富有表现力。简单来说，一句话：Kotlin 既可利用 Java 的优势，又比 Java 更简洁。

此外，Kotlin 程序还可直接编译生成 JavaScript 代码——Kotlin 程序既可编译成前端 JavaScript 代码，用于实现网页的 DOM 操作，实现前端编程；也可编译成后端 JavaScript 代码，与服务端技术（如 Node.js）交互。

不得不说的一点是，目前 Android 已推荐使用 Kotlin 作为官方开发语言，这意味着 Kotlin 将会在 Android 开发中大放异彩，这也是笔者决定向读者介绍这门语言的重要原因之一。

1.1 Kotlin 语言简介

1.1.1 服务端的 Kotlin

Kotlin 程序可以编译成 Java 字节码文件，字节码文件可以直接在 JVM 上运行，因此 Kotlin 非常适合开发后端应用程序。Kotlin 与现有的 Java 语言包能保持完全兼容，这意味着 Kotlin 不是一门简单的语言，它完全可以利用 Java 领域现有的各种技术框架，如 Spring、Hibernate、MyBatis、Lucene 等，因此 Java 开发者非常容易过渡到使用 Kotlin。

总结来看，Kotlin 的显著优势有如下几点。

- **简洁性**：这是我们选择使用 Kotlin 的最大动力。Kotlin 具有大量现代编程语言的简洁性和便捷性，因此 Kotlin 被誉为 Android 平台的 Swift。
- **兼容性**：Kotlin 完全兼容 Java，因此 Kotlin 既是一门新的语言，也不是一门“全新”的语言，Kotlin 可以自由使用 Java 领域的无数框架和库。因此，开发者既可保持熟悉的技术栈，又可获得现代编程语言的优势。
- **迁移性**：Kotlin 支持大型项目从 Java 向 Kotlin 逐步迁移——项目主体部分继续使用 Java，新开发的功能可使用 Kotlin 编写，也可逐步使用 Kotlin 代替部分老旧的 Java 代码。

短期内，Kotlin 不会对 Java 造成巨大的冲击，但 Kotlin 简洁、优雅的语法可以对 Java 形成良好的补充，开发者可根据需要自由选择 Java 或 Kotlin，最终都会生成字节码文件，运行于 JVM 平台上。

1.1.2 使用 Kotlin 开发 Android 应用

Google 官方推荐使用 Kotlin 作为 Android 开发语言，证明了 Kotlin 非常适合开发 Android 应用。使用 Kotlin 开发 Android 应用可充分利用 Kotlin 的简洁性和便捷性。

Kotlin 完全兼容 JDK 1.6，因此保证了基于 Kotlin 开发的 Android 应用完全可以在较老的 Android 设备上运行。

对于广大的 Android 应用开发者而言，大部分时候都是与 Android 应用程序框架层交互的，调用 Android 应用程序框架层的 API，而 Kotlin 可以自由调用 Java 的各种类库，因此使用 Kotlin 调用 Android 应用程序框架层来开发应用程序甚至无须额外学习，开发者可以无缝地过渡为使用 Kotlin 开发。

对于性能方面来说，Kotlin 编译的字节码与 Java 原生字节码极为相似。随着 Kotlin 对内联函数的支持，使用 Lambda 表达式的代码通常比用 Java 写的代码运行得更快。

正是基于以上两点主要优势，Google 官方推荐使用 Kotlin 作为 Android 开发语言。

1.1.3　Kotlin 用于 JavaScript

Kotlin 程序还可以编译成 JavaScript 代码，Kotlin 程序会生成遵守 ECMAScript 规范的 JavaScript 代码。当选择生成 JavaScript 目标时，不仅会包括开发者自己写的 Kotlin 代码，也会包括 Kotlin 附带的标准库，它们都会转换为 JavaScript。

Kotlin 既可生成前端使用的 JavaScript 代码，也可生成后端使用的 JavaScript 代码。

当生成前端 JavaScript 代码时，Kotlin 可实现如下功能。

- Kotlin 提供了大量 API 来操作 DOM（文档对象模型），允许通过 DOM API 来动态创建和更新页面。
- Kotlin 也提供了支持 WebGL 的 API，因此可以在网页上用 WebGL 创建图形元素。
- Kotlin 也可使用现有的前端库和框架，如 jQuery 和 ReactJS 等。

当生成后端 JavaScript 代码时，Kotlin 完全可以与后端 JavaScript（如 Node.js）进行交互。

本书重点介绍 Kotlin 生成 JVM 字节码，这种 Kotlin 程序完全可以兼容 Java 程序，这也是 Kotlin 开发 Android 应用的基础。

1.2　使用命令行编译、运行 Kotlin

下面先介绍使用命令行环境编译、运行 Kotlin。与其他编程语言类似，Kotlin 同样需要下载并安装对应的 SDK，接下来即可使用该 SDK 提供的命令来编译、运行 Kotlin 程序。

1.2.1　下载和安装 Kotlin 的 SDK

下载和安装 Kotlin SDK 请按如下步骤进行。

① 登录 https://github.com/JetBrains/kotlin/releases，看到 Kotlin SDK 的最新发布版本，即可看到如图 1.1 所示的页面。本书成书之时，Kotlin SDK 的最新版本是 1.1.4-2，本书所有的案例也是基于该版本 SDK 的。

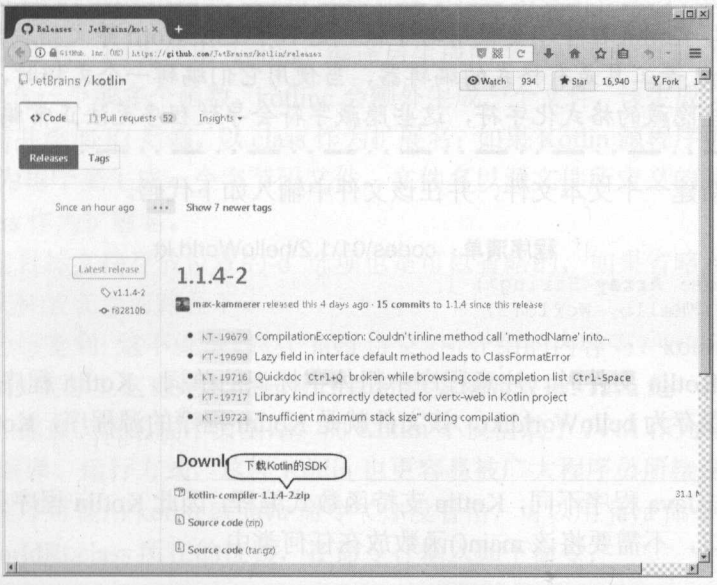


图 1.1　下载 Kotlin SDK 的页面

② 单击如图 1.1 所示页面中的链接，即可下载得到一个 kotlin-compiler-1.1.4-2.zip 压缩包文件。

③ 将该压缩包文件解压缩到任意目录下，即可看到如下文件路径。

- bin：该路径下存放了 Kotlin SDK 的各种工具命令，常用的 kotlinc、kotlin 等命令就放在该路径下。
- lib：该目录下包含了 Kotlin 的各种工具 JAR 包。
- license：存放与 Kotlin 项目相关的各种授权文档。

在上面路径中，bin 是一个非常有用的路径，在这个路径下包含了编译和运行 Kotlin 程序的 kotlinc 和 kotlin 两个命令。除此之外，还包含了 kotlinc-jvm、kotlinc-js 等工具命令。其中 kotlinc 和 kotlinc-jvm 是一样的，都用于将 Kotlin 程序编译成适用于 JVM 的字节码文件，而 kotlinc-js 则用于将 Kotlin 程序编译成 JavaScript 代码。

④ 将解压缩路径下的 bin 目录添加到系统的 PATH 环境变量中，这样即可保证操作系统能通过 PATH 环境变量找到 kotlinc、kotlin 等工具命令。

⑤ 将解压缩路径下的 lib 目录中的 kotlin-stdlib.jar 文件（该文件与 kotlin-runtime.jar 文件相同，都代表了 Kotlin 的运行时环境库）添加到 CLASSPATH 环境变量中，这样即可保证运行 Kotlin 程序时能正常加载 Kotlin 的运行时环境库。



提示：

本书假设读者已有最基本的 Java 编程基础，因此书中不会逐步介绍环境变量配置这些基础内容。如果读者需要学习环境变量配置这些基础知识，则建议参考《疯狂 Java 讲义》一书。

1.2.2 第一个 Kotlin 程序

编辑 Kotlin 源代码可以使用任何无格式的文本编辑器，在 Windows 操作系统上可使用记事本（NotePad）、EditPlus 等程序，在 Linux 平台上可使用 VI 工具等。



提示：

编写 Kotlin 程序不要使用写字板，更不可使用 Word 等文档编辑器。因为写字板、Word 等工具是有格式的编辑器，当使用它们编辑一个文档时，这个文档中会包含一些隐藏的格式化字符，这些隐藏字符会导致程序无法正常编译、运行。

在记事本中新建一个文本文件，并在该文件中输入如下代码。

程序清单：codes\01\1.2\helloWorld.kt

```
fun main(args: Array<String>) {
    println("Hello, World!")
}
```

编辑上面的 Kotlin 文件时，注意程序中粗体字标识的单词，Kotlin 程序严格区分大小写。将上面文本文件保存为 helloWorld.kt，该文件就是 Kotlin 程序的源程序，Kotlin 源程序的文件名要求以.kt 结尾。

Kotlin 程序与 Java 程序不同，Kotlin 支持函数式编程，因此 Kotlin 程序只需要一个 main()函数作为程序入口，不需要将该 main()函数放在任何类中。



提示：

仔细看一下上面的 main()函数，不难发现这个 main 函数其实就是 Java 主类中的 main()方法的变体。只不过在 Kotlin 语言中，函数也是一等公民，因此函数可以独立存在，而 Java 主类中的 main()方法必须放在类中声明。Java 主类中的 main()方法需要声明一个 String[] args 形参，而 Kotlin 的 mian()函数同样声明了 args: Array<String>形参——它们都是一个字符串数组类型的形参，这表明它们的本质是一样的，只是形式不同而已。至于 Kotlin 主函数前面的 fun 关键字，它专门用于声明函数；Swift 语言还使用 func 关键字声明函数呢，其实没有什么特别的，习惯不同而已。

编写好 Kotlin 程序的源代码后，接下来就应该编译该 Kotlin 源文件来生成字节码文件了。

1.2.3 编译、运行 Kotlin 程序

编译 Kotlin 程序需要使用 kotlinc 或 kotlinc-jvm 命令（两个命令完全一样），因为前面已经把 kotlinc 命令所在的路径添加到了系统的 PATH 环境变量中，因此现在可以使用 kotlinc 命令来编译 Kotlin 程序。

对于初学者而言，先掌握 kotlinc 命令的如下用法：

```
kotlinc -d destdir srcFile
```

在上面命令中，-d destdir 是 kotlinc 命令的选项，用以指定编译生成的字节码文件的存放路径，destdir 只需是本地磁盘上的一个有效路径即可；而 srcFile 是 Kotlin 源文件所在的位置，这个位置既可以是绝对路径，也可以是相对路径。

通常，总是将所生成的字节码文件放在当前路径下，当前路径可以用一点（.）来表示。在命令行窗口进入 helloWorld.kt 文件所在的路径，在该路径下输入如下命令：

```
kotlinc -d . helloWorld.kt
```

运行该命令后，在该路径下生成一个 HelloWorldKt.class 文件。

与 javac 编译器类似，使用 kotlinc 编译文件只需要指定存放目标文件的位置即可，无须指定字节码文件的文件名。因为使用 kotlinc 编译后生成的字节码文件有默认的文件名：如果 Kotlin 源程序中包含一个或多个函数，kotlinc 会额外生成一个文件，该生成文件的文件名是源文件名首字母大写并添加 Kt 后缀，以.class 作为扩展名；如果 Kotlin 源程序中包含一个或多个类，kotlinc 则会为每个类生成一个字节码文件，文件名以源文件所定义的各个类的类名作为主文件名，以.class 作为扩展名。

事实上，指定目标文件存放位置的-d 选项也是可以省略的，如果省略该选项，则意味着将生成的字节码文件放在当前路径下。

可能有读者会感觉到：这不就是《疯狂 Java 讲义》所介绍的内容吗？kotlinc 的用法与 javac 的用法何其相似啊！实际上这也是 Kotlin 设计者希望实现的效果：Java 是一门如此深入人心、影响深远的语言，拥有广泛的程序员基础，而 Kotlin 不仅借助于 JVM 作为运行平台，而且保持和 Java 相似的编译、运行方式，这样 Kotlin 也更容易被广大程序员所接受。

运行 Kotlin 程序可使用 kotlin 或 java 命令（你没看错，可以用 java 命令），启动命令行窗口，进入 HelloWorldKt.class 所在的位置，在命令行输入如下命令：

```
java HelloWorldKt
```

运行上面命令，将看到如下输出：

```
Hello World!
```

这表明 Kotlin 程序运行成功。

当然，Kotlin 也专门提供了一个 kotlin 命令来运行 Kotlin 程序，因此在命令行输入如下命令：

```
kotlin HelloWorldKt
```

运行上面命令，同样可以看到如下输出：

```
Hello World!
```

1.3 使用 IntelliJ IDEA 编译、运行 Kotlin

Kotlin 本身就是 JetBrains 开发的，而 IntelliJ IDEA 则是 JetBrains 自家开发的 IDE 工具，因此 IntelliJ IDEA 必然支持 Kotlin 语言。

IntelliJ IDEA 是一个商业版软件，使用该软件的商业版是要收费的，但 JetBrains 还为 IntelliJ IDEA 提供了一个免费的社区版，因此本书会基于免费的社区版 IntelliJ IDEA 进行介绍。

登录 https://www.jetbrains.com，下载并安装最新的社区版 IntelliJ IDEA。

通过 IntelliJ IDEA 新建一个支持 Kotlin 的 Java 项目，请按如下步骤进行。

① 单击 IntelliJ IDEA 的“File”→“New”→“Project...”新建一个 Java 项目，并为新建的项目勾选“Kotlin/JVM”复选框，如图 1.2 所示。

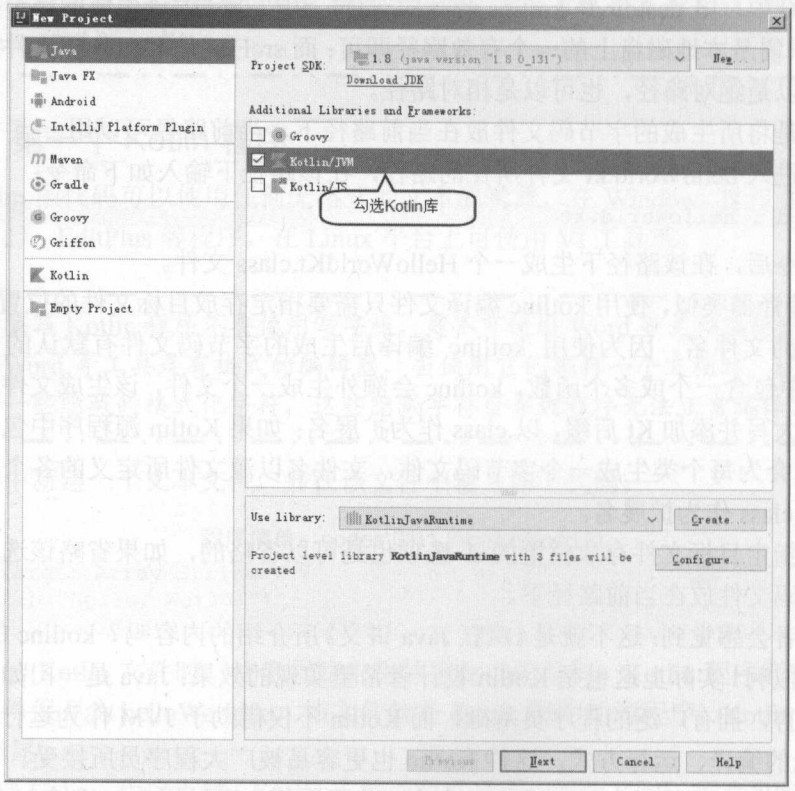


图 1.2 新建支持 Kotlin 的 Java 项目

② 让 Java 项目支持 Kotlin 的关键就是勾选“Kotlin/JVM”复选框，项目创建完成后可看到如图 1.3 所示的项目结构。

其实让 Java 项目支持 Kotlin 也很简单，无非就是需要两个条件：

- 该项目“知道”使用 kotlinc 编译器来编译 Kotlin 程序。
- 该项目包含了 Kotlin 的运行时环境。

正如从图 1.3 所看到的，上面项目中包含了 kotlin-stdlib.jar，这就是 Kotlin 的运行时环境库（即本章 1.2 节让大家添加到 CLASSPATH 环境变量中的 JAR 包），而 kotlin-reflect.jar 是 Kotlin 反射才需要的 JAR 包，kotlin-stdlib-jre7.jar、kotlin-stdlib-jre8.jar 是 Kotlin 为支持 Java 7、Java 8 提供的运行时环境库。此外，由于 IntelliJ IDEA 本身就是 JetBrains 自家的 IDE 工具，让它集成一个 kotlinc 编译器是顺理成章的事——这就是该项目能支持 Kotlin 程序的根本所在。

③ 右键单击图 1.3 所示项目结构中的 src 目录，在弹出的快捷菜单中选择“New”→“Kotlin File/Class”菜单项，如图 1.4 所示。

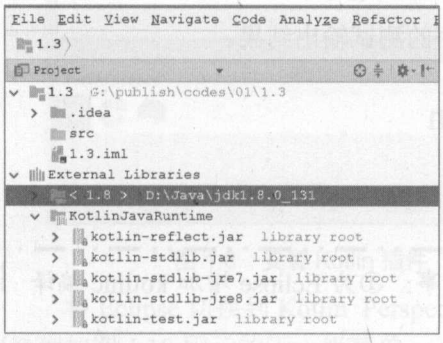


图 1.3　支持 Kotlin 的 Java 项目

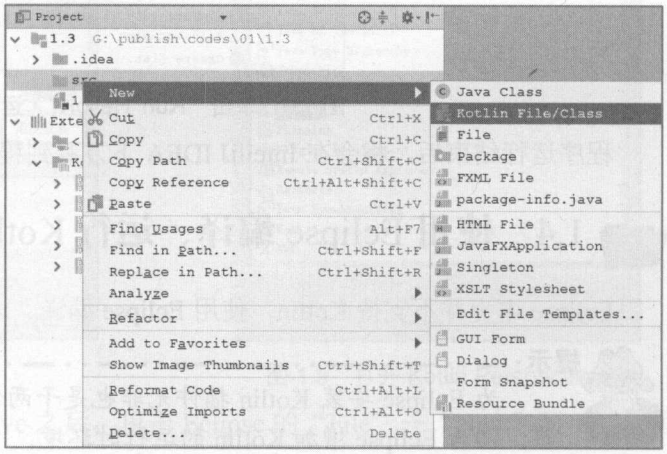


图 1.4　新建 Kotlin 文件

④ 系统弹出如图 1.5 所示的对话框，在该对话框中可以选择 Kotlin 文件的类型。

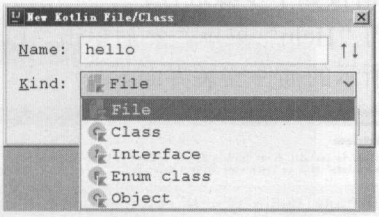


图 1.5　选择 Kotlin 文件的类型

对于熟悉 Java 的读者来说，对类、接口和枚举这些文件类型太熟悉了，它们都是 Java 程序员的老朋友。但在这里不打算选择这些老朋友，而是直接选择 File 类型——这表明将会新建一个普通的 Kotlin 文件。正如前面所介绍的，Kotlin 支持函数式编程，函数也是 Kotlin 的一等公民，因此此处将直接在 Kotlin 程序中新建主函数作为程序入口。

⑤ 在 hello.kt 文件中添加一个主函数，IntelliJ IDEA 提供了一个快速完成此操作的模板，只需输入 main，然后按 Tab 键，即可在编辑界面中看到添加了 main()函数。

在 main()函数中添加一行简单的输出语句用于测试。

⑥ 在 hello.kt 的编辑界面中单击鼠标右键，系统弹出如图 1.6 所示的快捷菜单，单击该菜单中的“Run 'HelloKt'”菜单项即可运行该程序。

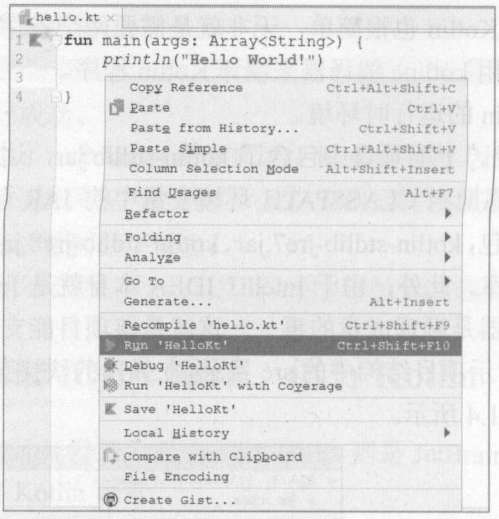


图 1.6 单击“Run 'HelloKt'”运行 Kotlin 程序

程序运行结束后，将会在 IntelliJ IDEA 下方看到程序的测试输出结果。

1.4 使用 Eclipse 编译、运行 Kotlin

Eclipse 本身并不支持 Kotlin，使用 Eclipse 编译、运行 Kotlin 需要安装插件。



提示：

为 Eclipse 安装 Kotlin 插件无非也是干两件事：①为 Eclipse 增加 kotlinc 编译器；②为 Eclipse 增加 Kotlin 的运行时环境。

由于本书假设读者已有一定的 Java 基础，因此关于 Eclipse 的安装步骤本书就不介绍了。为 Eclipse 安装 Kotlin 插件请按如下步骤进行。

① 单击 Eclipse 主菜单中的“Help”菜单，然后单击“Eclipse Marketplace”菜单项，即可看到 Eclipse 弹出的如图 1.7 所示的插件安装界面。

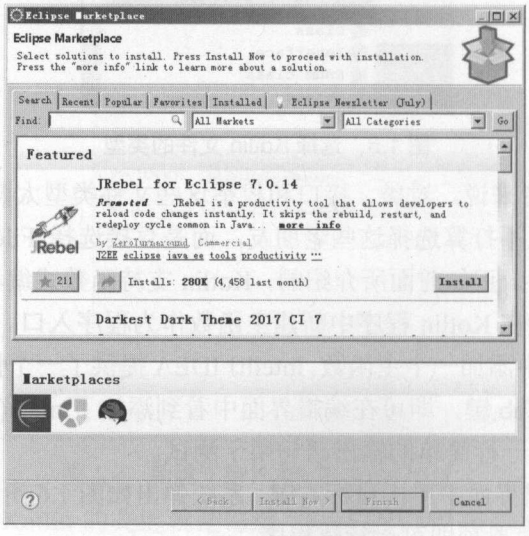


图 1.7 Eclipse 插件安装界面

② 在 Eclipse 插件安装界面上方的 Find 文本框内输入“Kotlin”，然后单击文本框右边的“搜索”图标（放大镜图标），即可看到如图 1.8 所示的界面。

③ 单击右侧的“Install”按钮，Eclipse 将会弹出一个“同意授权”的对话框，在该对话框中选中“I accept...”单选钮同意授权，然后单击“Finish”按钮。接下来 Eclipse 将会自动下载、安装 Kotlin 插件，插件安装完成后需要重启 Eclipse。

为 Eclipse 成功安装 Kotlin 插件之后，即可通过 Eclipse 的“Window”→“Perspective”→“Open Perspective ”→ “Other...”来打开 Kotlin 的 Perspective，如图 1.9 所示。

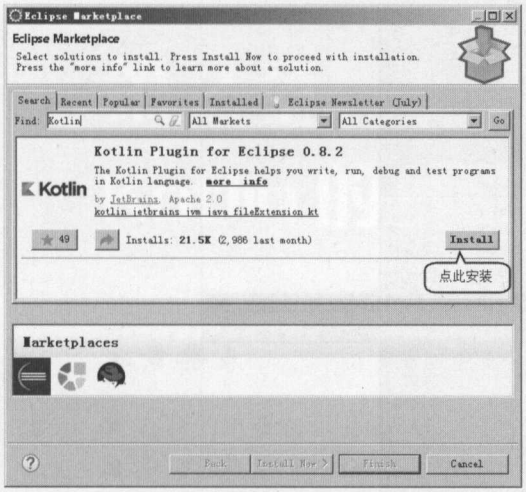


图 1.8　安装 Kotlin 插件

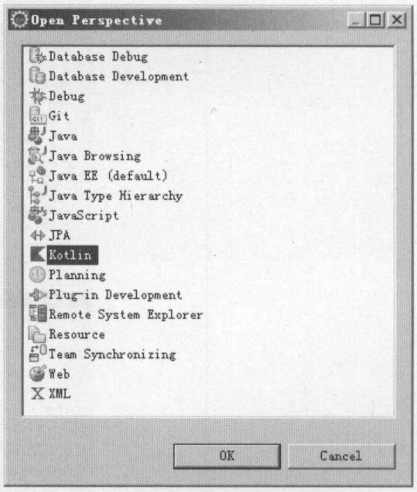


图 1.9　打开 Kotlin Perspective

将 Eclipse 切换到 Kotlin Perspective 之后，单击 Eclipse 的“File”→“New”菜单，即可看到如图 1.10 所示的下一级菜单。

在下一级菜单中单击 “Kotlin Project”菜单项即可新建 Kotlin 项目，项目建立完成后可以看到如图 1.11 所示的项目结构。

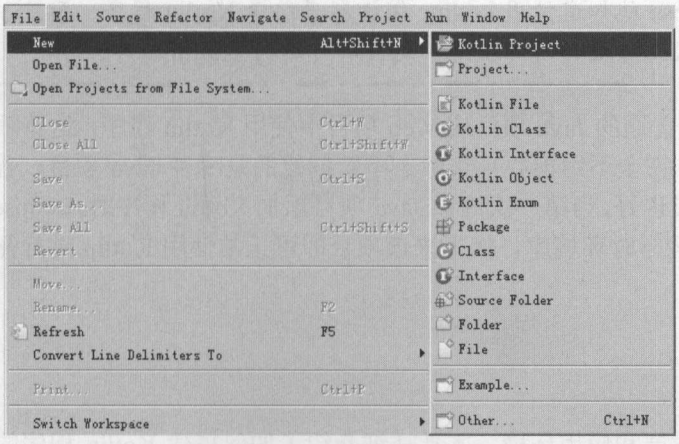


图 1.10　关于新建 Kotlin 的相关菜单

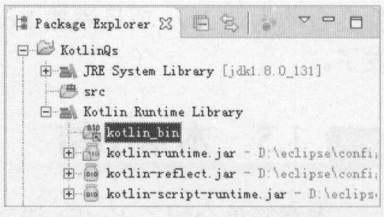


图 1.11　Kotlin 项目结构

从图 1.11 可以看出，Kotlin 项目就是额外增加了 Kotlin 运行时环境库，该运行时环境库也就是添加了 kotlin-runtime.jar（与 kotlin-stdlib.jar 相同）和 kotlin-reflect.jar 两个运行时环境 JAR 包，这和前面使用 IntelliJ IDEA 所创建的 Kotlin 项目的本质完全相同。

接下来再次通过图 1.10 所示的下一级菜单创建 Kotlin 文件，编辑该文件，在该文件中添

加 Kotlin 程序所需的 main()函数，并在该 main()函数中添加输出语句。

在 Kotlin 程序的编辑界面中单击鼠标右键，系统弹出如图 1.12 所示的快捷菜单，单击该菜单中的“Run As” →“1 Kotlin Application”菜单项即可运行该程序。

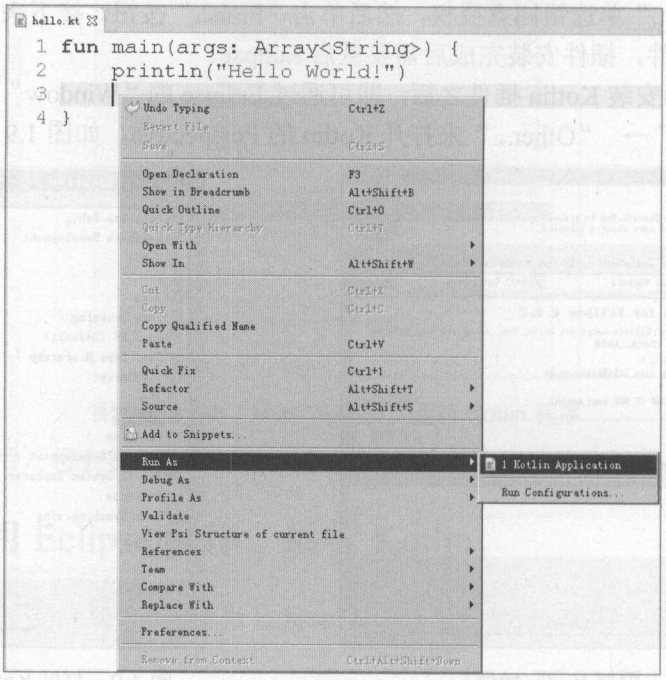


图 1.12 运行 Kotlin 程序

程序运行结束后，将会在 Eclipse 下方看到程序的测试输出结果。



提示：

使用 Eclipse 运行 Java 程序与运行 Kotlin 程序并没有什么区别。为 Eclipse 安装了 Kotlin 插件之后，Eclipse 就知道使用 kotlinc 编译器来编译 Kotlin 程序，而且 Eclipse 也为该项目添加了 Kotlin 运行时环境库，这样即可运行 Kotlin 程序。

可能有读者感到担心：如果要在普通的 Java 项目或 Web 项目中使用 Kotlin 程序，这些项目没有 Kotlin 运行时环境库怎么办？完全不用担心！实际上不管创建的是哪种 Java 项目，普通的 Java SE 项目也好，Java Web 项目也好，当第一次为该 Java 项目添加 Kotlin 程序时，Eclipse 就会自动为该 Java 项目添加 Kotlin 运行时环境库，接下来该项目即可正常使用 Kotlin 进行开发了。

1.5 本章小结

本章简单介绍了 Kotlin 语言的发展和应用环境。本章详细介绍了如何搭建 Kotlin 程序的开发环境，包括下载和安装 Kotlin 的 SDK，如何通过命令行编译、运行 Kotlin 程序。此外，本章还介绍了如何使用 IntelliJ IDEA、Eclipse 等 IDE 工具来编译、运行 Kotlin 程序。当然，也可使用其他 IDE 工具来编译、运行 Kotlin 程序，基本方式与此差别不大。学习本章的重点就是搭建 Kotlin 程序开发、运行环境，并成功运行第一个 Kotlin 程序。

CHAPTER

2

第2章 Kotlin的基础类型

本章要点

- 注释的重要性和用途
- 单行注释语法和多行注释语法
- 文档注释
- Kotlin 的标识符规则
- 变量和常量
- 整型的功能和用法
- 浮点型的功能和用法
- 字符型的功能和用法
- 布尔型的功能和用法
- 可空类型
- 先判断可空类型的非空，再调用
- 可空类型的安全调用
- Kotlin 字符串及常见操作
- 类型别名

与 Java 相同，Kotlin 也是一门强类型的语言。强类型包含两方面的含义：①所有的变量必须先声明、后使用；②指定类型的变量只能接受类型与之匹配的值。强类型的语言可以在编译过程中发现源代码的错误，从而保证程序更加健壮。

本章作为学习 Kotlin 编程的基础，将会向读者详细介绍变量和常量的语法，以及变量和常量的基本用法。除此之外，本章会重点介绍 Kotlin 的整型、浮点型、布尔型、字符、字符串等基础类型，并通过示例介绍各种数据类型的功能、用法，以及相关注意事项。

2.1 注释

Kotlin的注释与Java基本保持一致，Kotlin注释同样支持单行注释、多行注释和文档注释。

2.1.1 单行注释和多行注释

单行注释就是在程序中注释一行代码，在 Kotlin 语言中，将双斜线（//）放在需要注释的内容之前，就可以了；多行注释是指一次性地将程序中的多行代码注释掉，在 Kotlin 语言中，使用“/*”和“*/”将程序中需要注释的内容包含起来，其中“/*”表示注释开始，而“*/”表示注释结束。

需要指出的是，Java 语言的多行注释不支持嵌套，而 Kotlin 语言的多行注释支持嵌套，这样用起来就更方便了。

下面代码中增加了单行注释和多行注释。

程序清单：codes\02\2.1\comment.kt

```
/*
这里面的内容全部是多行注释
Kotlin 语言真的很简单
*/
fun main(args: Array<String>) {
    // 这是一行简单的注释
    println("Hello World!")
    // print("这行代码被注释了，将不会被编译、执行!")
    /* 这是第一个多行注释的开头
    /* 这是第二个被嵌套的多行注释 */
    这是第一个多行注释的结尾 */
}
```

注意上面代码中最后一段粗体字注释，这段多行注释中再次嵌套了多行注释，这在 Java 语言中是不允许的，但 Kotlin 是允许的。可见，Kotlin 的多行注释可以嵌套。也就是说，在/*...*/多行注释块内，可以再次使用/*...*/添加多行注释。当使用多行注释嵌套另一个多行注释时，只要注意先插入第二个注释块的结束标记，再插入第一个注释块的结束标记即可。

除此之外，添加注释也是调试程序的一种重要方式——如果觉得某段代码可能有问题，则可以先把这段代码注释起来，让编译器忽略这段代码，再次编译、运行这个程序，如果可以正常编译、运行，则可以说明错误就是由这段代码引起的，这样就缩小了错误所在的范围，有利于排错；如果依然出现相同的错误，则可以说明这个错误不是由这段代码引起的，同样也缩小了错误所在的范围。

2.1.2 文档注释

Kotlin 的文档注释和 Java 相同，同样使用/**和*/来进行文档注释，中间部分全部都是文

档注释，会被提取到 API 文档中。

下面先编写一个 KdocTest 类，这个类里包含了对类、方法的文档注释。

程序清单：codes\02\2.1\KdocTest.kt

```
package lee
/**
 * Description:
 * 网站: <a href="http://www.crazyit.org">疯狂 Java 联盟</a>
 * Copyright (C), 2001-2018, Yeeku.H.Lee
 * This program is protected by copyright laws.
 * Program Name:
 * Date:
 * @author Yeeku.H.Lee kongyeeku@163.com
 * @version 1.0
 * @property name the name of this group.
 * @constructor Creates an empty group.
 */
public class KDocTest {
    /**
     * 一个做加法的简单方法
     * @param a 第一个加数
     * @param b 第二个加数
     * @return 两个数的和
     */
    public fun add(a: Int, b: Int): Int {
        return a + b
    }
}
```

再编写一个 test 文件，该文件包含了一个公共函数。

程序清单：codes\02\2.1\test.kt

```
package yeeku

/**
 * 打印消息的简单函数
 * @param msg 要显示的消息
 */
fun showMsg(msg: String){
    println(msg);
}
```

上面 Kotlin 程序中的粗体字标识部分就是文档注释。编写好上面的 Kotlin 程序后，接下来需要使用 Dokka 工具来生成 API 文档。

注意：

一定要为上面的源程序指定包名，否则 Dokka 工具不会提取文档注释来生成 API 文档。

Kotlin 的 SDK 默认不包含 Dokka 工具，因此开发者需要自行下载。登录 https://github.com/Kotlin/dokka 即可下载该工具，下载完成后得到一个 dokka-fatjar.jar 文件，该文件是一个可执行的 JAR 包，执行该 JAR 包即可启动 Dokka 工具。

Dokka 工具的基础用法如下：

```
java -jar dokka-fatjar.jar <源文件或目录> <参数>
```

Dokka 工具可对源文件、包生成 API 文档。在上面的语法格式中，Kotlin 源文件可以支持

通配符，例如，使用*.kt 来代表当前路径下所有的 Kotlin 源文件。Dokka 的常用参数有如下几个。

- -output <directory>：该参数指定所生成的 API 文档的存放路径。该参数的默认值为 out，也就是将所生成的 API 文档放在 out 目录下。
- -format：该参数指定所生成的 API 文档的格式。

使用 format 参数指定 API 文档格式时支持如下常用格式选项。

- html：生成最简的 HTML 格式的 API 文档。
- javadoc：生成符合 javadoc 格式的 API 文档。
- html-as-java：基本与 html 相似，但会使用 Java 语法。

除此之外，Dokka 工具还包含了大量其他选项，读者可以通过 https://github.com/Kotlin/dokka 来了解它们的更多使用细节。

在命令行窗口执行如下命令，为刚刚编写的两个 Kotlin 程序生成 API 文档：

```
java -jar dokka-fatjar.jar KDocTest.kt test.kt
```

在 KDocTest.kt 和 test.kt 所在路径下执行上面命令，可以看到生成 API 文档的提示信息。进入 KDocTest.kt 和 test.kt 所在路径，可以看到一个 out 文件夹，该文件夹下的内容就是刚刚生成的 API 文档。进入 out/doc 路径下，打开 index.html 文件，将看到如图 2.1 所示的页面。

单击如图 2.1 所示 API 文档上方的 lee 或 yeeku 包，进入该包的详细页面，即可看到该包所包含的函数和类。

如果开发者希望生成类似于 javadoc 格式的 API 文档（一般不建议），则可以在使用 Dokka 工具时添加-format 选项，并为该选项指定 javadoc 选项值。例如，输入如下命令：

```
java -jar dokka-fatjar.jar KDocTest.kt test.kt -format javadoc
```

再次打开 out/doc 目录下的 index.html 页面，即可看到如图 2.2 所示的 API 文档。

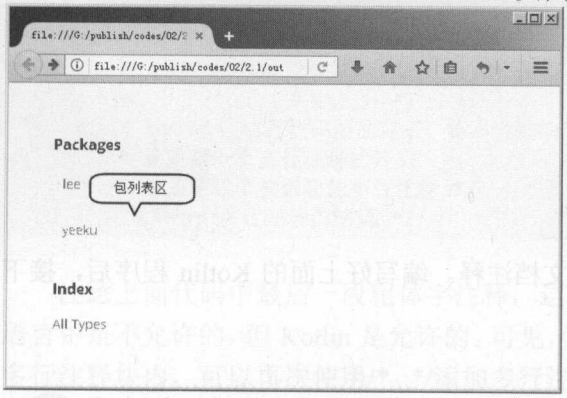


图 2.1 所生成的 API 文档

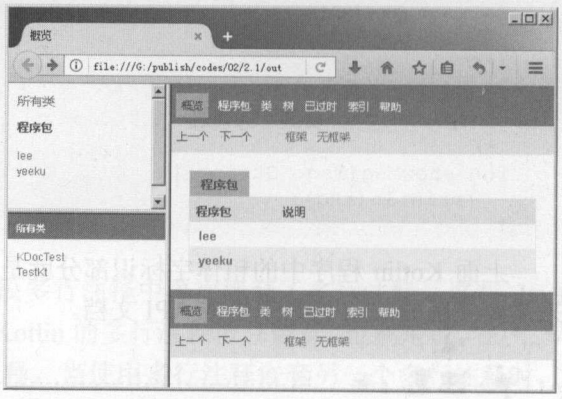


图 2.2 使用 Dokka 工具生成 javadoc 格式的文档

如图 2.2 所示的 API 文档格式是我们熟悉的 javadoc 格式。一般来说，并不建议使用 Dokka 工具来生成 javadoc 格式的 API 文档，因为等我们真正习惯了 Dokka 默认的文档格式之后，就会觉得该文档格式也很方便易用。

2.2 变量

无论使用什么语言编程，总要处理数据，处理数据就需要使用变量来保存数据。形象地看，变量就像一个个小容器，用于“盛装”程序中的数据。常量同样也用于“盛装”程序中的数据，

常量与变量的区别是：常量一旦保存某个数据之后，该数据就不能发生改变；但变量保存的数据则可以多次发生改变，只要程序对变量重新赋值即可。

2.2.1 分隔符

Kotlin 语言里的分号（;）、花括号（{}）、方括号([])、圆括号（()）、空格、圆点（.）都具有特殊的分隔作用，因此被统称为分隔符。

1. 分号

与 Java 不同，Kotlin 并不强制要求每条语句必须以分号结尾，当然也可以使用分号表示语句结束。正如从前面程序所看到的，如果单独一行内只有一条语句，那么 Kotlin 允许这条语句不以分号结尾。

但如果打算在同一行内书写多条独立的语句，则前面语句需要使用分号表示结束。例如如下语句：

```
var name = "fkit"; println(name)
```

Kotlin 允许一条语句可以跨多行。例如，如下语句都是正确的。

```
var str = "fkit"
// 表达式可以跨多行
str +=
    ".org"
// 调用方法时可以跨多行
var hasFk = str
    .startsWith("fk")
var st = "fkit"
// 访问属性时也可以跨多行
var len = st
    .length
```

值得指出的是，一个字符串、变量名不能跨多行。例如，下面的 Kotlin 语句是错误的。

```
// 字符串不能跨多行
var a = "dddddd
    xxxxxxx"
// 变量名不能跨多行
var na
    me = "李刚"
```

不仅如此，虽然 Kotlin 语法允许在一行内书写多条语句，但从程序的可读性角度来看，应该避免在一行内书写多条语句。

注意：

Kotlin 语句可以跨多行书写，但字符串和变量名不能跨多行。虽然 Kotlin 语法允许在一行内书写多条语句，但从程序的可读性角度来看，应该避免在一行内书写多条语句。



2. 花括号

花括号的作用就是定义一个代码块。一个代码块指的就是“{”和“}”所包含的一段代码，代码块在逻辑上是一个整体。对 Kotlin 语言而言，类体部分、枚举需要放在花括号中定义，方法体也必须放在花括号中定义。除此之外，条件语句中的条件执行体和循环语句中的循环体通常也放在代码块里。

花括号一般是成对出现的，有一个“{”则必然有一个“}”，反之亦然。后面将会详细介绍花括号分隔符的用法，此处不再赘述。

3. 方括号

Kotlin 的方括号是一个索引运算符，其主要作用是用于访问数组元素、List 集合元素和 Map 集合元素。方括号通常紧跟数组变量名、List 变量名或 Map 变量名，而在方括号里指定希望访问的数组元素、List 元素的索引或 Map 元素的 key。

例如如下代码：

```
// 下面代码试图对名为 a 的数组的第四个元素赋值
a[3] = 3
// 下面代码试图对名为 scores 的字典中 key 为 Kotlin 的元素赋值
scores["Kotlin"] = 92
```

关于方括号分隔符的用法，后面还会有更进一步的介绍，此处不再赘述。

4. 圆括号

圆括号是一个功能非常丰富的分隔符，在定义函数、方法时必须使用圆括号来包含所有的形参声明，在调用函数时也必须使用圆括号来传入实参值。不仅如此，圆括号还可以将表达式中某个部分括成一个整体，保证这个部分优先计算。

关于圆括号分隔符在后面还有更进一步的介绍，此处不再赘述。

5. 空格

Kotlin 语言使用空格分隔一条语句的不同部分。Kotlin 语言是一门格式自由的语言，所以空格几乎可以出现在 Kotlin 程序的任何地方，也可以出现任意多个空格，但不要使用空格把一个变量名隔开成两个，这将导致程序出错。

Kotlin 语言中的空格包含空格符（Space）、制表符（Tab）和回车符（Enter）等。

除此之外，Kotlin 源程序还会使用空格合理地缩进代码，从而提供更好的可读性。

6. 圆点

圆点（.）通常用作类、结构体、枚举、实例和它的成员（包括属性和方法）之间的分隔符，表明调用某个类或某个实例的指定成员。关于圆点分隔符的用法，后面还会有更进一步的介绍，此处不再赘述。

2.2.2 标识符规则

所谓标识符，就是用于给程序中的变量、类、枚举、函数等命名的名字。标识符的长度没有限制。

Kotlin 语言是区分大小写的，因此 abc 和 Abc 是两个不同的标识符。

在使用标识符时，需要注意如下规则。

- 标识符可以由字符、数字和下画线（_）组成，但不能以数字开头。此处的字符并不局限于 26 个英文字母，可以包含中文字符、日文字符等。
- 标识符不能是 Kotlin 的硬关键字，在特定上下文中不能使用软关键字和修饰符关键字，但可以包含关键字。
- 标识符不能包含空格。
- 标识符只能包含下画线（_），不能包含@、#等特殊字符。

2.2.3 Kotlin的关键字

在Kotlin语言中有一些具有特殊用途的单词被称为关键字（keyword），当定义标识符时，不要让标识符与关键字相同，否则将引起错误。例如，如下代码将会引起错误：

```
// 下面代码试图定义一个名为 class 的变量，但 class 是关键字，不能作为标识符
var class = 2
```

Kotlin的关键字可分为3类。

- **硬关键字**：这些关键字无论在什么情况下都不能用作标识符。
- **软关键字**：这些关键字可以在它们不起作用的上下文中用作标识符。
- **修饰符关键字**：这些关键字也可以在代码中用作标识符。

Kotlin的硬关键字包括：

- as——用于做类型转换或为import语句指定别名。
- as?——类型安全的类型转换运算符。
- break——中断循环。
- class——声明类。
- continue——忽略本次循环剩下的语句，重新开始下一次循环。
- do——用于do while循环。
- else——在if分支中使用。
- false——在Boolean类型中表示假的直接量。
- for——用于for循环。
- fun——声明函数。
- if——在if分支中使用。
- in——在for循环中使用；in还可作为双目运算符，检查一个值是否处于区间或集合内；in也可在when表达式中使用；in还可用于修饰泛型参数，表明该泛型参数支持逆变。
- !in——可作为双目运算符in的反义词；!in也可在when表达式中使用。
- is——用于做类型检查（类似Java的instanceof）或在when表达式中使用。
- !is——用于做类型检查（is的反义词）或在when表达式中使用。
- null——代表空的直接量。
- object——用于声明对象表达式或定义命名对象。
- package——用于为当前文件指定包。
- return——声明函数的返回。
- super——用于引用父类实现的方法或属性，或者在子类构造器中调用父类构造器。
- this——代表当前类的对象或在构造器中调用当前类的其他构造器。
- throw——用于抛出异常。
- true——在Boolean类型中表示真的直接量。
- try——开始异常处理。
- typealias——用于定义类型别名。
- val——声明只读属性或变量。
- var——声明可变属性或变量。
- when——用于when表达式。
- while——用于while循环或do while循环。

Kotlin 的软关键字包括：

- by——用于将接口或祖先类的实现代理给其他对象。
- catch——在异常处理中用于捕捉异常。
- constructor——用于声明构造器。
- delegate——用于指定该注解修饰委托属性存储其委托实例的字段。
- dynamic——主要用于在 Kotlin/JavaScript 中引用一个动态类型。
- field——用于指定该注解修饰属性的幕后字段。
- file——用于指定该注解修饰该源文件本身。
- finally——异常处理中的 finally 块。
- get——用于声明属性的 getter 方法，或者用于指定该注解修饰属性的 getter 方法。
- import——用于导包。
- init——用于声明初始化块。
- param——用于指定该注解修饰构造器参数。
- property——用于指定该注解修饰整个属性（这种目标的注解对 Java 不可见，因为 Java 并没有真正的属性）。
- receiveris——用于指定该注解修饰扩展方法或扩展属性的接收者。
- set——用于声明属性的 setter 方法，或者用于指定该注解修饰属性的 setter 方法。
- setparam——用于指定该注解修饰 setter 方法的参数。
- where——用于为泛型参数增加限制。

Kotlin 的修饰符关键字包括：

- abstract——用于修饰抽象类或抽象成员。
- annotation——用于修饰一个注解类。
- companion——用于声明一个伴生对象。
- const——用于声明编译时常量。
- crossinline——用于禁止在传给内联函数的 Lambda 表达式中执行非局部返回。
- data——用于声明数据类。
- enum——用于声明枚举。
- external——用于声明某个方法不由 Kotlin 实现（与 Java 的 native 相似）。
- final——用于禁止被重写。
- infix——声明该函数能以双目运算符的格式执行。
- inline——用于声明内联函数，Lambda 表达式可在内联函数中执行局部返回。
- inner——用于声明内部类，内部类可以访问外部类的实例。
- internal——用于表示被修饰的声明只能在当前模块内可见。
- lateinit——用于修饰一个 non-null 属性，用于指定该属性可在构造器以外的地方完成初始化。
- noinline——用于禁止内联函数中个别 Lambda 表达式被内联化。
- open——用于修饰类，表示该类可派生子类；或者用于修饰成员，表示该成员可以被重写。
- out——用于修饰泛型参数，表明该泛型参数支持协变。
- override——用于声明重写父类的成员。
- private——private 访问权限。
- protected——protected 访问权限。

- public——public 访问权限。
- reified——用于修饰内联函数中的泛型形参，接下来在该函数中就可像使用普通类型一样使用该类型参数。
- sealed——用于声明一个密封类。
- suspend——用于标识一个函数后 Lambda 表达式可作为暂停。
- tailrec——用于修饰一个函数可作为尾随递归函数使用。
- vararg——用于修饰形参，表明该参数是个数可变的形参。

2.2.4 声明变量

Kotlin 是强类型的语言，Kotlin 要求所有的变量必须先声明、后使用，声明变量时必须显式或隐式指定变量的类型，类型限制了一个变量能被赋的值，也限制了一个表达式可以产生的值，还限制了在这些值上可以进行的操作，并确定了这些操作的含义。

声明变量需要使用 var 或 val，如下所示：

```
var|val 变量名 [:类型] [= 初始值]
```

其中使用 var 声明的变量是值可变的（可被多次赋值），使用 val 声明的变量是值不可变的（不能被重新赋值）。

在上面声明变量的语法格式中，程序要么通过“:类型”的形式显式指定该变量的类型，要么为该变量指定初始值——Kotlin 编译器将会根据该初始值确定变量的类型，不能声明变量时既不指定变量类型，也不指定变量初始值。声明变量时可以既显式指定变量的类型，也指定该变量的初始值，但显式指定的变量类型必须与初始值的类型保持一致。

例如如下代码。

程序清单：codes\02\2.2\VarTest.kt

```
public fun main(args: Array<String>) {
    // 声明变量时显式指定类型
    var b : Int
    // 声明变量时指定初始值，编译器会根据初始值确定该变量的类型为 String
    var name = "crazyit.org"
    b = 20 // b 的类型是 Int（整型），赋值正确
    name = "fkit.org" // name 的类型为 String，赋值正确
    name = 12 // name 的类型为 String，但 12 为 Int，赋值错误
    // 声明变量时既显式指定类型，也指定初始值
    // 显式指定的类型与初始值的类型一致，声明变量正确
    var age: Int = 25
    age = 12 // age 的类型是 Int，赋值正确
    // 声明变量时既显式指定类型，也指定初始值
    // 显式指定的类型与初始值的类型不一致，声明变量失败
    var sun : String = 500
    val book = "疯狂 Kotlin 讲义"
    book = "a" // 使用 val 声明的变量是不可变变量，不能被重新赋值
}
```

对上面程序中 5 行粗体字代码的解释如下。

- 第 1 行粗体字代码声明变量时只显式指定了变量的类型，并未指定初始值，因此变量 b 只能接受 Int 类型的值。
- 第 2 行粗体字代码声明 name 变量时并未显式指定变量的类型，但程序为该变量指定的初始值为字符串"crazyit.org"，因此编译器可以确定该变量的类型为 String，该变量只能接受字符串类型的值。

- 第 3 行粗体字代码声明 age 变量时既指定了该变量的类型为 Int，也指定了该变量的初始值为 25，程序指定的变量类型与变量初始值的类型是一致的，因此程序正确。
- 第 4 行粗体字代码声明 sun 变量时指定了该变量的类型为 String，但程序又尝试为该变量指定 500 作为初始值，这样显式指定的变量类型与变量的初始值类型并不一致，因此这行代码是错误的。
- 第 5 行粗体字代码使用 val 声明了不可变变量，因此该变量不允许被重新赋值，所以接下来为 book 重新赋值的代码错误。

使用 val 声明的不可变变量其实相当于常量，这意味着它的值一旦被初始化之后，将不可以被重新赋值。根据常量所在的位置不同，Kotlin 的常量分为两种。

- **局部范围的常量**：这种常量允许在声明时不指定初始值，只要在第一次使用之前指定初始值即可。
- **类的常量属性**：这种常量属性既可以在声明时指定初始值，也可以在类或结构体的构造器中指定初始值。

程序清单：codes\02\2.2\ValTest.kt

```
fun main(args: Array<String>) {
    // 定义常量，没有显式指定类型，编译器根据初始值确定常量的类型
    val maxAge = 120
    // 定义常量时，既显式指定了常量的类型，也指定了常量的初始值
    val eduName : String = "疯狂软件教育"
    // 常量不允许重新赋值，因此下面代码是错误的
    // maxAge = 12
    // 同时定义多个变量
    // 局部范围的常量，声明时不指定初始值
    val herName: String
}
```

对上面程序中 3 行粗体字代码的解释如下。

- 第 1 行粗体字代码声明常量时只指定了常量的初始值，编译器将根据初始值来确定该常量的类型。
- 第 2 行粗体字代码声明 eduName 常量时既显式指定了该常量的类型，也指定了该常量的初始值。
- 第 3 行粗体字代码使用 val 声明了一个 String 类型的常量，但并未指定初始值——这是因为局部范围的常量，并不要求在定义时指定初始值，只要在第一次使用之前指定初始值即可。

需要指出的是，由于 Kotlin 程序编译的字节码必须遵守 JVM 规范，因此，如果直接在 Kotlin 程序中定义变量、函数，kotlinc 将会自动生成一个名为“文件名首字母大写+Kt”的类，并将变量转换为该类的静态的 getter、setter 方法（其中 val 声明的只有 getter 方法），函数则转换为该类的静态方法。

注意：

当使用 kotlinc 编译 Kotlin 程序所包含的函数、变量时，这些函数、变量都会转换成该文件对应类的静态方法。



还有一点需要说明的是，由于 kotlinc 会为包含函数、变量的 Kotlin 程序生成额外的类，这就要求不能在该包下重复定义同名的类。例如，我们定义了一个名为 ligang.kt 的 Kotlin 程序，且该程序中包含了函数或变量，那么 kotlinc 就会自动生成 LigangKt 类，因此就不能在该

包下重复定义 LigangKt 类。

2.3 整型

与 Java 类似，Kotlin 也提供了 4 种整型。

- Byte：Byte 型整数在内存中通常占 8 位，表数范围是-128~127。兼容 Java 的 byte 和 Byte 类型。
- Short：Short 型整数在内存中占 16 位，表数范围是-32768(-2^{15})~32767(2^{15}-1)。兼容 Java 的 short 和 Short 类型。
- Int：Int 型整数在内存中占 32 位，表数范围是-2147483648(-2^{31})~-2147483647(2^{31}-1)。兼容 Java 的 int 和 Integer 类型。
- Long：Long 型整数在内存中占 64 位，表数范围是-2^{63}~2^{63}-1。兼容 Java 的 long 和 Long 类型。

当程序直接给出一个较大的整数时，该整数默认可能就是 Long 型，如果将这个整数赋值给 Int、Short 或 Byte 型，编译器将会报错。

例如如下代码。

程序清单：codes\02\2.3\IntVarTest.kt

```
fun main(args: Array<String>) {
    // 下面代码是正确的
    var a:Int = 56
    // 下面代码需要隐式地将 2999999999 转换为 Int 型使用，因此编译器将会报错
    var bigValue : Int = 2999999999
    // 下面代码是正确的
    var bigValue2: Long = 2999999999
    println(bigValue)
    println(bigValue2)
    println(Short.MIN_VALUE)
    println(Short.MAX_VALUE)
}
```

上面程序中的粗体字代码将 2999999999 作为 Int 型变量的初始值，但 2999999999 的值处于 Long 型的取值范围内，因此 2999999999 将被当成 Long 型使用。故而上面这行粗体字代码报错。



提示：

一般来说，开发者应该尽量使用 Int 型，这样可以提高代码的一致性和可复用性。

由于 Int 型是 Kotlin 最常用的整数类型，因此，如果声明一个常量或变量时没有指定数据类型，只是简单地指定了其初始值为整数值，那么 Kotlin 会自动判断该变量的类型为 Int。

Kotlin 的整型与 Java 不同，Kotlin 的整型不是基本类型，而是引用类型（大致相当于 Java 的包装类），Byte、Short、Int、Long 都继承了 Number 类型，因此它们都可调用方法、访问属性。

程序可通过访问不同整数类型的 MIN_VALUE 和 MAX_VALUE 属性来获取对应类型的最大值和最小值。例如如下代码（程序清单同上）：

```
println(Short.MIN_VALUE)
println(Short.MAX_VALUE)
```

有一点需要提前说明的是，Kotlin 是 null 安全的语言，因此 Byte、Short、Int、Long 型变量都不能接受 null 值，如果要存储 null 值，则应该使用 Byte?、Short?、Int?、Long?类型。例如如下代码（程序清单同上）：

```
// Int 型变量不支持 null 值，所以下面代码是错误的
var notNull: Int = null
// Int?相当于支持 null 值的 Int 型，所以下面代码是正确的
var nullable: Int? = null
```

由此可见，Kotlin 语言允许在已有数据类型后添加“?”，添加“?”后的数据类型相当于对原有类型进行了扩展，带“?”的数据类型可支持被赋予 null 值。

此外，整数类型添加“?”后缀与不加后缀还有一个区别——普通类型的整型变量将会映射成 Java 的基本类型；带“?”后缀的整型变量将会映射成基本类型的包装类。举例来说，Kotlin 程序中 Int 类型的变量将会映射成 Java 的 int 基本类型，但 Int?类型的变量则会自动映射成 Java 的 Integer 类型。例如如下代码（程序清单同上）：

```
var pm1: Int = 200; // pm1 的类型是 Java 的 int 类型
var pm2: Int = 200; // pm2 的类型是 Java 的 int 类型
println(pm1 === pm2); // 基本类型比较，输出 true
var ob1: Int? = 200; // ob1 的类型是 Java 的 Integer 类型
var ob2: Int? = 200; // ob2 的类型是 Java 的 Integer 类型
println(ob1 === ob2); // 引用类型比较，输出 false
```

上面代码中由于 pm1、pm2 两个变量都映射成 Java 的基本类型，因此，即使用“===”比较相等（该运算符要求两个变量引用同一个对象）也返回 true；但 ob1、ob2 两个变量都映射成 Java 的 Integer 对象，此时用 ob1 和 ob2 不再引用同一个对象，因此它们用“===”比较返回 false。

注意：

如果将 ob1、ob2 的值改为小于-128 ~ 127 之间的整数，那么上面 ob1 和 ob2 用“===”比较依然是相等的，这是由 Java 的 Integer 类的特征决定的。关于这一点可参考《疯狂 Java 讲义》的 6.2 节。



Kotlin 的整数数值有 3 种表示方式。

- **十进制**：最普通的整数数值就是十进制的整数。
- **二进制**：以 0b 或 0B 开头的整数数值就是二进制的整数。
- **十六进制**：以 0x 或 0X 开头的整数数值就是十六进制的整数，其中 10~15 分别以 a~f（此处的 a~f 不区分大小写）来表示。

注意：

Kotlin 不支持八进制的整数。



下面代码片段分别使用了二进制、十六进制的整数。

程序清单：codes\02\2.3\HexTest.kt

```
fun main(args: Array<String>) {
    // 以 0b 或 0B 开头的整数数值是二进制的整数
    var binValue1 = 0b1010101
    var binValue2 = 0B10101110
    // 以 0x 或 0X 开头的整数数值是十六进制的整数
```

```
    var hexValue1 = 0x13
    var hexValue2 = 0XaF
    println("binValue1的值为：${binValue1}")
    println("binValue2的值为：${binValue2}")
    println("hexValue1的值为：${hexValue1}")
    println("hexValue2的值为：${hexValue2}")
}
```

上面代码中用到了字符串模板，也就是在字符串中嵌入${}的形式，在${}内可放入变量或表达式，Kotlin会将该变量或表达式的值“嵌入”该字符串内。

为了提高数值（包括浮点型）的可读性，Kotlin允许为数值（包括浮点型）增加下画线作为分隔符，也可以在数值前添加额外的零。这些下画线和零并不会影响数值本身。例如如下代码（程序清单同上）：

```
val oneMillion = 1_000_000
val price = 234_234_234 // price实际的值为234234234
val android = 1234_1234 // android实际的值为12341234
```

2.4 浮点型

浮点型数值可以包含小数部分，浮点型比整型的表数范围更大，可以存储比Long型更大或更小的数。Kotlin的浮点型有两种。

- Float：表示32位的浮点型，当精度要求不高时可以使用此种类型。
- Double：表示64位的双精度浮点型，当程序要求存储很大或者高精度的浮点数时使用这种类型。

Kotlin的浮点数有两种表示形式。

- **十进制数形式**：这种形式就是简单的浮点数，例如5.12、512.0、0.512等。浮点数必须包含一个小数点，否则会被当成整数类型处理。
- **科学计数形式**：例如5.12e2（即5.12×10^2）、5.12E2（也是5.12×10^2）等。

必须指出的是，只有浮点型的数值才可以使用科学计数形式表示。例如51200是一个Int型的值，但512E2则是浮点型的值。

如果声明一个常量或变量时没有指定数据类型，只是简单地指定了其初始值为浮点数，那么Kotlin会自动判断该变量的类型为Double。

除此之外，Kotlin还提供了3个特殊的浮点型数值：正无穷大、负无穷大和非数。例如，使用一个正数除以0.0将得到正无穷大数值，使用一个负数除以0.0将得到负无穷大数值，0.0除以0.0或对负数开方将得到一个非数。

需要指出的是，所有的正无穷大数值都相等，所有的负无穷大数值都相等；而非数不与任何数值相等，甚至和非数自己都不相等。

注意：

只有浮点数除以0.0才可以得到正无穷大数值或负无穷大数值，但如果使用整型值除以整数0将会引起编译错误：division by zero（除0错误）。



下面程序示范了上面介绍的关于浮点数的各个知识点。

程序清单：codes\02\2.4\FloatTest.kt

```
fun main(args: Array<String>) {
```

```
    var af1 = 5.2345556f;
    // 下面将看到 af1 的值已经发生了改变
    println("af1 的值为:${af1}")
    // 声明 af2 是 Float 类型，但 25.2345 默认是 Double 类型，因此下面代码编译时报错
    var af2: Float = 25.2345
    // f1 的类型被推断为 Double
    var f1 = 5.12e2
    println("f1 的值为:${f1}")
    var a = 0.0
    // 5.0 除以 0.0 将出现正无穷大数值
    println("5.0/a 的值为: ${5.0 / a}")
    // 所有的正无穷大数值都相等，所以下面将会输出 true
    println(5.0 / a == 50000 / 0.0)
    // -5.0 除以 0.0 将出现负无穷大数值
    println("-5.0/a 的值为: ${-5.0 / a}")
    // 所有的负无穷大数值都相等，所以下面将会输出 true
    println(-5.0 / a == -50000 / 0.0)
    // 0.0 除以 0.0 将出现非数
    var nan : Double = a / a
    println("a/a 的值为: ${nan}")
    // 非数与自己都不相等，所以下面将会输出 false
    println(nan == nan)
}
```

2.5 字符型

字符型通常用于表示单个的字符，字符型值必须使用单引号（'）括起来。Kotlin 语言使用 16 位的 Unicode 字符集作为编码方式，而 Unicode 被设计成支持世界上所有书面语言的字符，包括中文字符，因此 Kotlin 程序支持各种语言的字符。

字符型值有如下 3 种表示形式。

- 直接通过单个字符来指定字符型值，例如'A'、'9'和'0'等。
- 通过转义字符表示特殊字符型值，例如'\n'、'\t'等。
- 直接使用 Unicode 值来表示字符型值，格式是'\uXXXX'，其中 XXXX 代表一个十六进制的整数。

Kotlin 语言中常用的转义字符如表 2.1 所示。

表 2.1 Kotlin 语言中常用的转义字符

转义字符	说 明	Unicode 表示方式
\b	退格符	\u0008
\n	换行符	\u000a
\r	回车符	\u000d
\t	制表符	\u0009
\"	双引号	\u0022
\'	单引号	\u0027
\\	反斜线	\u005c

字符型值也可以采用十六进制编码方式来表示，范围是'\u0000'~'\uFFFF'，一共可以表示 65536 个字符，其中前 256 个（'\u0000'~'\u00FF'）字符和 ASCII 码中的字符完全重合。

与 Java 不同的是，Kotlin 的 Char 型变量不能当成整数值使用，Char 型变量或表达式不能

赋值给整型变量，整型变量或表达式也不能赋值给 Char 型变量。简单来说，Kotlin 的 Char 型就是简简单单的字符型，不能直接当成整型使用。

下面程序简单示范了字符型变量的用法。

程序清单：codes\02\2.5\CharTest.kt

```
fun main(args: Array<String>) {
    // 直接指定单个字符作为字符值
    val aChar: Char = 'a'
    // 使用转义字符来作为字符值
    val enterChar: Char = '\r'
    // 使用 Unicode 编码值来指定字符值
    val ch: Char = '\u9999'
    // 将输出一个'香'字符
    println(ch)
    // 定义一个'疯'字符值
    var zhong: Char = '疯'
    // 将 Char 型变量当成 Int 型处理会报错
    var zhongValue: Int = zhong
}
```

2.6 数值型之间的类型转换

Kotlin 是一门强类型的语言，Kotlin 编译器对类型检查非常严格，这种严格保证了 Kotlin 程序的健壮性，因此 Kotlin 不同类型的值经常需要进行相互转换，数值型之间的变量和值也可以相互转换。

2.6.1 整型之间的转换

不同整型的变量能支持的表数范围是不同的，比如 Byte 类型的变量或常量只能接受-128~127 之间的整数，Short 类型的变量或常量只能接受-32768~32767 之间的整数。如果数值超出了变量或常量所支持的表数范围，编译器就会报错。例如如下代码：

```
// Short 类型支持的表数范围为-32768~32767，所以下面代码报错
var negative: Short = 40000
// Byte 类型支持的表数范围为-128~127，所以下面代码报错
var big: Byte = 128
```

简单来说，Kotlin 与 Java 不同，Kotlin 不支持取值范围小的数据类型隐式转换为取值范围大的类型。

由于不同整型支持的表数范围存在差异，因此进行类型转换时必须注意选择合适的类型。Kotlin 为所有数值类型都提供了如下方法进行转换。

- toByte()：转换为 Byte 类型。
- toShort()：转换为 Short 类型。
- toInt()：转换为 Int 类型。
- toLong()：转换为 Long 类型。
- toFloat()：转换为 Float 类型。
- toDouble()：转换为 Double 类型。
- toChar()：转换为 Char 类型。

Kotlin 要求不同整型的变量或值之间必须进行显式转换。例如如下代码。

程序清单：codes\02\2.6\IntConvert.kt

```
fun main(args: Array<String>) {
    var bookPrice : Byte = 79
    var itemPrice : Short = 120
    // bookPrice 是 Byte 类型，但变量 a 是 Short 类型，因此下面代码错误
//    var a: Short = bookPrice
    // 显式将 bookPrice 强制转换为 Short 类型
    var a: Short = bookPrice.toShort()  // ①
    var b: Byte = itemPrice.toByte()
    println("a: ${a}, b: ${b}")
    val amount = 233
    // 将 Int 型变量转换为 Byte 类型，发生溢出
    val byteAmount: Byte = amount.toByte() // ②
    println(byteAmount)
}
```

上面程序中第一行粗体字代码试图将 bookPrice（Byte 类型的变量）赋值给变量 a，但由于变量 a 是 Short 类型，虽然它们都是整型，但 Kotlin 依然不允许直接赋值，因此这行代码无法通过编译。

程序中①号粗体字代码先将 bookPrice 强制转换为 Short 类型，这样即可将转换后的结果赋值给 Short 类型的变量 a；接下来的代码则调用 itemPrice 的 toByte()方法将 Short 类型的变量转换为 Byte 类型。

与 Java 类似的是，把取值范围大的变量或表达式强制转换为取值范围小的类型时，可能发生溢出。例如上面程序中②号粗体字代码所示。上面程序还把 233 强制转换为 Byte 类型整数，从而变成了-23，这就是典型的溢出。图 2.3 示范了这个转换过程。

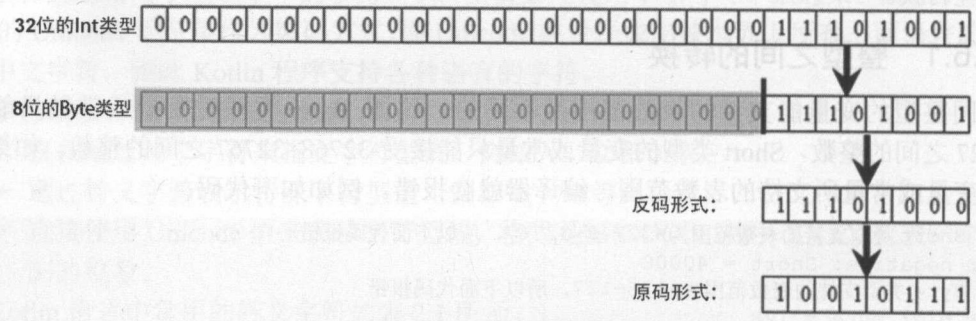


图 2.3 Int 类型向 Byte 类型强制转换

从图 2.3 可以看出，32 位 Int 类型的 233 在内存中的存储示意如图中最上面所示，强制转换为 8 位的 Byte 类型，则需要截断前面的 24 位，只保留右边的 8 位，其中最左边的 1 是符号位，此处表明这是一个负数，负数在计算机里是以补码形式存在的，因此还需要换算成原码。

将补码减 1 得到反码形式，再将反码取反就可以得到原码。

最后的二进制原码为 10010111，这个 Byte 类型的值为-(16 + 4 + 2 + 1)，也就是-23。

从图 2.3 很容易看出，当试图把表数范围大的类型强制转换为表数范围小的类型时，必须格外小心，因为非常容易引起信息丢失。

虽然 Kotlin 缺乏隐式转换，但 Kotlin 在表达式中可以自动转换，这种转换是基于上下文推断出来的，而且算术运算会有重载做适当转换。例如如下代码（程序清单同上）：

```
// 算术表达式中的 bookPrice、itemPrice 会自动提升为 Int 类型
var total = bookPrice + itemPrice  // ③
println("total 的值为:${total}")
// 可以看到 total 映射的 Java 类型为 int
```

```
println("total 的类型为:${total.javaClass}")
// 下面表达式中的 bookPrice 强制转换为 Long 类型，因此整个表达式类型为 Long
val tot = bookPrice.toLong() + itemPrice.toByte()  // ④
println("total 的值为:${tot}")
// 可以看到 total 映射的 Java 类型为 long
println("total 的类型为:${tot.javaClass}")
```

上面③号粗体字代码将 bookPrice（Byte 类型）和 itemPrice（Short 类型）相加，Kotlin 并不需要对它们进行强制转换，Kotlin 将会自动把它们提升到 Int 类型之后再进行计算（与 Java 表达式对所有整型的处理规则相同），因此整个表达式计算得到的结果是 Int 类型。

上面④号粗体字代码先将 bookPrice 的类型强制转换为 Long 类型，这样整个表达式中最高等级的操作数是 Long 类型，因此整个表达式计算得到的结果也是 Long 类型。



提示：

上面程序中使用了变量的 javaClass 属性，该属性来自 Any 类型（Any 类型是 Kotlin 所有类型的根父类），javaClass 属性用于获取指定变量对应的 Java 类型（大致相当于 Java 反射中的 getClass()方法）。

Kotlin 虽然不允许直接将 Char 型值当成整数值使用，也不允许将整数值直接当成 Char 型值使用，但 Kotlin 依然可调用数值型的 toChar()方法将数值型变量或表达式转换为 Char 类型。例如，下面程序示范了如何生成一个 6 位的随机字符串，这个程序中用到了后面的循环控制，不理解循环的读者可以参考后面章节中的介绍。

程序清单：codes\02\2.6\RandomStr.kt

```
fun main(args: Array<String>) {
    // 定义一个空字符串
    var result = "";
    // 进行 6 次循环
    for(i in 0..5) {
        // 生成一个 97~122 之间的 Int 类型整数
        val intVal = (Math.random() * 26 + 97).toInt();
        // 将 intValue 强制转换为 Char 类型后连接到 result 后面
        result = result + intVal.toChar();
    }
    // 输出随机字符串
    println(result);
}
```

此外，Char 型值虽然不能被当成整数进行算术运算，但 Kotlin 为 Char 类型提供了加、减运算支持，其计算规则如下。

- **Char 型值加、减一个整型值**：Kotlin 会先将该 Char 型值对应的字符编码进行加、减该整数，然后将计算结果转换为 Char 型值。
- **两个 Char 型值进行相减**：Kotlin 将用两个 Char 型值对应的字符编码进行减法运算，最后返回 Int 类型的值。两个 Char 型值不能相加。

例如如下程序。

程序清单：codes\02\2.6\CharAdd.kt

```
fun main(args: Array<String>) {
    var c1 = 'i'
    var c2 = 'k'
    println(c1 + 4); // 输出 m
    println(c1 - 4); // 输出 e
```

```
    println((c1 - c2)); // 输出-2
}
```

2.6.2 浮点型与整型之间的转换

Kotlin 的 Float、Double 之间需要进行显式转换，浮点型与整型之间也需要进行显式转换——其转换过程与前面介绍的整型之间的转换过程基本相似。例如如下程序。

程序清单：codes\02\2.6\FloatConvert.kt

```
fun main(args: Array<String>) {
    var width: Float = 2.3f
    var height: Double = 4.5
    // width 必须显式强制转换为 Double 之后，才能赋值给变量 a
    var a: Double = width.toDouble()
    println("a 的值为: ${a}")
    // 将 height 强制转换为 Float 之后再进行计算，整个表达式的类型是 Float
    // 因此 area1 的类型也被推断为 Float
    var area1 = width * height.toFloat()
    // 表达式中的 height 是 Double 类型，它是等级最高的运算数
    // 因此整个表达式的类型是 Double，area2 的类型也被推断为 Double
    var area2 = width * height
    val multi: Int = 5
    // 因此 totalHeight1 的类型也被推断为 Double
    var totalHeight1 = height * multi  // ①
    // 将 height 强制转换为 Int 类型后进行计算，整个表达式的类型是 Int
    // 因此 totalHeight2 的类型也被推断为 Int
    var totalHeight2 = height.toInt() * multi  // ②
}
```

上面程序中第一行粗体字代码希望将 Float 类型的 width 变量赋值给 Double 类型的变量 a，因此程序必须先进行类型转换。接下来程序试图将 width 和 height 相乘的积赋值给 area1、area2，如果我们希望 area1 变量的类型是 Float，这就需要将 height 强制转换为 Float 类型，这样整个表达式中的 width、height 都是 Float 类型，整个表达式的计算结果也是 Float 类型。area2 则将被推断为 Double 类型。

上面程序中①号粗体字代码计算 multi、height 的乘积，由于 height 的类型是 Double，因此整个表达式的类型是 Double。所以 totalHeight1 的类型也是 Double。

程序中②号粗体字代码先将 height 强制转换为 Int 类型，然后与 multi（也是 Int 类型）变量相乘，这样代码也是正确的。

将 Double 类型或 Float 类型的值、变量或常量强制转换为整型时，浮点值的小数部分会被截断，例如 4.75 将会变成 4、-2.9 将会被截断为-2。对于②号粗体字代码而言，height 被强制转换为 Int 类型，此时小数部分将会截断，因此 height 将会被截断为 4。所以计算得到的 totalHeight2 的值为 20。

通过上面的介绍不难发现，当进行类型转换时，应该尽量向表数范围大的数据类型转换，这样程序会更加安全，比如前面介绍的 Byte 向 Short 转换、Int 向 Double 转换，而反过来转换则可能导致溢出。Kotlin 语言的各种数值型的表数范围由小到大的顺序为：Byte→Short→Int→Long→Float→Double。

2.6.3 表达式类型的自动提升

当一个算术表达式中包含多个数值型的值时，整个算术表达式的数据类型将发生自动提升。Kotlin 定义了与 Java 基本相似的自动提升规则。

- 所有的 Byte、Short 类型将被提升到 Int 类型。
- 整个算术表达式的数据类型自动提升到与表达式中最高等级操作数同样的类型。操作数的等级排列如图 2.4 所示，位于箭头右边的类型等级高于位于箭头左边的类型等级。

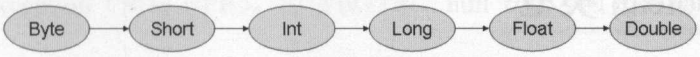


图 2.4 操作数的等级排列

下面程序示范了一个典型的错误。

程序清单：codes\02\2.6\AutoPromote.kt

```
// 定义一个 Short 类型变量
var sValue: Short = 5
// 表达式中的 sValue 将自动提升到 Int 类型，则右边的表达式类型为 Int
// 将一个 Int 类型值赋给 Short 类型变量将发生错误
sValue = sValue - 2
```

上面的“sValue - 2”表达式的类型将被提升到 Int 类型，这样就把右边的 Int 类型值赋给左边的 Short 类型变量，从而引起错误。

下面是表达式类型自动提升的正确示例代码（程序清单同上）。

```
var b: Byte = 40
var c: Short = 97
var i: Int = 23
var d: Double = .314
// 右边表达式中最高等级的操作数为 d（Double 类型）
// 则右边表达式的类型为 Double，result 将会推断为 Double 类型
val result = b + c + i * d
// 将输出 144.222
println(result)
```

必须指出，表达式的类型将严格保持和表达式中最高等级操作数相同的类型。下面代码中的两个 Int 类型整数进行除法运算，即使无法除尽，也将得到一个 Int 类型结果（程序清单同上）。

```
var iVal: Int = 3
// 右边表达式中的两个操作数都是 Int 类型，故右边表达式的类型为 Int
// 虽然 23/3 不能除尽，但依然得到一个 Int 类型整数
val intResult = 23 / iVal;
println(intResult) // 将输出 7
```

从上面程序中可以看出，当两个整数进行除法运算时，如果不能整除，得到的结果将是把小数部分截断取整后的整数。

如果表达式中包含了字符串，则又是另一番情形了。因为，当把加号（+）放在字符串和数值之间时，这个加号是一个字符串连接运算符，而不是进行加法运算。看如下代码：

```
// 输出字符串 Hello!a7
println("Hello!" + 'a' + 7)
// 输出字符串 hHello!
println('a' + 7 + "Hello!"))
```

对于第一个表达式“"Hello!" + 'a' + 7”，先进行“"Hello!" + 'a'”运算，把'a'转换成字符串，拼接成字符串 Hello!a；接着进行“"Hello!a" + 7”运算，这也是一个字符串连接运算，得到的结果是 Hello!a7。对于第二个表达式，先进行“'a' + 7”加法运算，此时是对 Char 型值做加法

运算，将会得到'a'后面的第 7 个字符（字符编码加 7），因此会得到'h'；接着进行“'h' + "Hello!"”运算，'h'会自动转换成字符串，将变成两个字符串的连接运算，从而得到 hHello!。

2.7 Boolean 类型

布尔型只有一个 Boolean 类型，用于表示逻辑上的“真”或“假”。与 Java 类似，Kotlin 的 Boolean 类型的值只能是 true 或 false，不能用 0 或者非 0 来代表。其他数据类型的值也不能转换成 Boolean 类型。

例如，下面代码定义了两个 Boolean 类型的变量，并指定初始值。

程序清单：codes\02\2.7\BooleanTest.kt

```
fun main(args: Array<String>) {
    // 定义 b1 的值为 true
    var b1 : Boolean = true
    // 定义 b2 的值为 false
    var b2 : Boolean = false
}
```

字符串"true"和"false"不会直接转换成 Boolean 类型，但 Boolean 类型的变量同样可以“插值”到字符串中。看下面代码（程序清单同上）：

```
// 将 Boolean 类型的变量插值到字符串中
var str: String = "${b1}代表真"
println(str)
```

与前面介绍的数值型、Char 类型类似的是，Boolean 类型的变量不能接受 null 值，Boolean? 类型的变量才能接受 null 值。Boolean 类型将直接映射为 Java 的 boolean 基本类型，但 Boolean? 类型将会映射成 boolean 的包装类：Boolean。

Boolean 类型的值或变量主要用作旗标来进行流程控制，在 Kotlin 语言中使用 Boolean 类型的变量或值控制的流程主要有如下几种。

- if 条件控制语句。
- while 循环控制语句。
- do while 循环控制语句。

如果程序在需要使用 Boolean 类型的地方使用了非 Boolean 值，Kotlin 的类型检查机制会报错。下面例子会报告一个编译错误。

```
var i = 1
if (i) {
    // 这个例子不能通过编译，会报错
}
```

下面例子则是正确的。

```
var i = 1
if (i == 1) {
    // 这个例子可以通过编译，不会报错
}
```

上面程序中第一个例子的 if 后面期望一个 Boolean 类型的值，但常量 i 是 Int 类型的，因此这个例子无法通过编译；在第二个例子中，i == 1 比较的结果是 Boolean 类型，因此这个例子可以通过编译检查。

2.8 null 安全

null 安全可以说是 Kotlin 语言对 Java 的重大改进之一，这样可以避免 Java 编程时令人恐惧的 NullPointerException（简称 NPE）。但话说回来，null 安全不过是各种现代编程语言玩剩下的东西。

2.8.1 非空类型和可空类型

下面先看一个简单的例子。

程序清单：codes\02\2.8\NullableTest.kt

```
fun main(args: Array<String>) {
    var str = "fkit"
    // 由于 str 转换为 Int 有可能失败，故 num 有可能没有值
    // 因此不能使用 Int 来声明 num 的类型
    var num: Int = str.toIntOrNull()
    var num: Int? = str.toIntOrNull()
    println(num)
}
```

对比上面两行粗体字代码：第一行代码声明 num 的类型为 Int，第二行代码声明 num 的类型为 Int?。程序中第一行粗体字代码无法通过编译，第二行粗体字代码能通过编译。其中 Int? 就是可空类型，这种类型的变量可接受 Int 值和 null；而 Int 类型的变量则只接受 Int 值，不能接受 null。由于 str 是一个 String 变量，当程序试图把 String 变量转换为 Int 值时，有可能转换成功（如果变量值是形如"123"的字符串），也有可能转换失败（本程序就转换失败了）。转换失败时，就无法成功返回 Int 值，此时将会返回 null，因此必须使用 Int?类型的变量来存储转换结果。

对于可能发生“值缺失”的情况，编译器会自动推断该变量的类型为可空类型。例如如下代码：

```
// 编译器推断 n 的类型为 Int?
var n = str.toIntOrNull()
```

需要指出的是，只有可空类型的变量或常量才能接受 null，非空类型的变量或常量不能接受 null。

Kotlin 对可空类型进行了限制：如果不加任何处理，可空类型不允许直接调用方法、访问属性。因此，通过可空类型与非空类型的区分，Kotlin 即可在程序中避免空指针异常。例如如下程序（程序清单同上）：

```
var aStr: String = "fkit"
var bStr: String? = "fkit"
aStr = null // 错误，aStr 不接受 null
bStr = null // 正确
// 编译通过，aStr 不可能为 null，运行时不可能导致 NPE
println(aStr.length)
// 编译不能通过，不可能导致 NPE
println(bStr.length)
```

上面代码中定义了 aStr 和 bStr 两个变量，其中 aStr 是非空的 String 类型，因此 aStr 变量不允许赋值为 null；而 bStr 是可空的 String 类型，bStr 变量允许被赋值为 null。程序可以直接调用 aStr 的方法或属性，但 aStr 变量的值不可能为 null，因此可以避免 NPE。Kotlin 对可空类型进行了限制，可空类型不允许直接调用方法或属性，因此程序不能直接调用 bStr 的方法或

属性，这样也可避免 NPE。

2.8.2 先判断后使用

可空类型的变量不允许直接调用方法或属性，但可以先判断该变量不为 null，然后再调用该变量的方法或属性。例如如下代码。

程序清单：codes\02\2.8\CheckNull.kt

```
fun main(args: Array<String>) {
    var b: String? = "fkit"
    // 先判断 b 不为 null，然后访问 b 的 length 属性
    var len = if (b != null) b.length else -1
    println("b 的长度:${len}")
    b = null
    // 先判断 b 不为 null，然后调用 b 的 length 属性
    if (b != null && b.length > 0) {
        // 访问 b 的 length 属性
        println(b.length)
    }
    else {
        println("空字符串")
    }
}
```

上面程序定义了 String?（可空类型）的变量 b，这样程序不能直接调用变量 b 的方法或属性。Kotlin 要求程序先判断 b 不为 null，接下来程序即可在该条件下调用 b 的方法或属性。

对于非空的 Boolean 类型而言，它可以接受 3 个值，即 true、false 或 null，因此对 Boolean? 类型变量进行判断时，需要使用 Boolean?变量显式与 true、false 值进行比较。例如如下代码：

```
fun main(args: Array<String>) {
    var b: Boolean? = null
    if( b == true ){
        println("为真")
    }
}
```

如果将 if 分支改为如下形式：

```
    if( b ){
        println("为真")
    }
```

编译器会报错：type mismatch: inferred type is Boolean? but Boolean was expected。这是因为 Kotlin 的 if 条件必须是 Boolean 类型，而 Boolean?与 Boolean 本质上是两种不同的类型，因此编译器会报错。

2.8.3 安全调用

安全调用在 Java 中已出现很多年了，只不过没有出现在 Java 官方语法中——如果读者熟悉 Spring EL，一定见过如下用法：

```
user?.dog?.name
```

上面表达式语言表示如果 user 不为 null，则返回 user 的 dog 属性；如果 dog 属性值不为 null，则继续获取 dog 属性值的 name 属性值。反过来，如果 user 为 null，或 user.dog 为 null，那么该表达式语言都不会导致 NPE，而是整个表达式返回 null——这就是 Spring EL 的安全调用。

Kotlin 的安全调用也完全与此类似，例如如下代码。

程序清单：codes\02\2.8\SafeCall.kt

```
fun main(args: Array<String>) {
    var b: String? = "fkit"
    println(b?.length) // 输出 4
    b = null
    println(b?.length) // 输出 null
}
```

上面程序中变量 b 的类型是 String?，因此程序使用了“?.”安全调用来访问 b 的 length 属性，当 b 为 null 时，程序也不会引发 NPE，而是返回 null。

与 Spring EL 类似的是，Kotlin 的安全调用也完全支持链式调用，就像 Spring EL 的用法一样：

```
user?.dog?.name
```

上面代码表示安全获取 user 的 dog 的 name 属性值，如果 user 或 user.dog 为 null，整个表达式将会返回 null。

此外，安全调用还可与 let 全局函数结合使用，例如如下代码（程序清单同上）：

```
// 定义一个元素可空的数组
val myArr: Array<String?> = arrayOf("fkit", "fkjava", null, "crazyit")
for (item in myArr) {
    // 当 item 不为 null 时才调用 let 函数
    item?.let { println(it) }
}
```

上面粗体字代码使用安全调用来调用 let 函数，这样只有当 item 元素不为 null 时才会执行 let 函数。上面程序调用 let 函数时传入一个 Lambda 表达式作为函数参数。

2.8.4 Elvis 运算

Elvis 运算也是一种小技巧，其实就是 if else 的简化写法。对比如下代码。

程序清单：codes\02\2.8\Elvis.kt

```
fun main(args: Array<String>) {
    var b: String? = "fkit"
    // 先判断 b 不为 null，然后访问 b 的 length 属性
    var len1 = if (b != null) b.length else -1
    println(len1);
    b = null
    // 使用 Elvis 运算符
    var len2 = b?.length ?: -1
    println(len2);
}
```

上面程序中第一行粗体字代码使用传统的 if 分支进行判断，当 b 不为 null 时返回 b.length，否则返回-1；第二行粗体字代码则使用“?:”运算符，该运算符就是 Elvis——它的含义是，如果“?:”左边的表达式不为 null，则返回左边表达式的值，否则返回“?:”右边表达式的值。

由此可见，“?:”其实就是 if 分支的简化写法。

此外，由于 Kotlin 的 return、throw 都属于表达式，因此它们也都可以用在“?:”运算符的右边。例如如下代码片段：

```
val data = ……
val email = data["email"] ?: throw IllegalArgumentException("没有指定 E-mail 信息!")
```

2.8.5 强制调用

强制调用是为NPE“爱好者”准备的——如果读者依然喜欢Java那种简单、粗暴的方式：不管变量是否为null，程序都直接调用该变量的方法或属性。Kotlin也为这种用法提供了支持，用“!!.”即可强制调用可空变量的方法或属性，这样强制调用可能引发NPE。例如如下代码。

程序清单：codes\02\2.8\ForceCall.kt

```
fun main(args: Array<String>) {
    var b: String? = "fkit"
    println(b!!.length) // 输出 4
    b = null
    println(b!!.length) // 引发空指针异常（NPE）
    // 定义一个元素可空的数组
    val myArr: Array<String?> = arrayOf("fkit", "fkjava", null, "crazyit")
    for (item in myArr) {
        // 当 item 不为 null 时才调用 let 函数
        item!!.let { println(it) }
    }
}
```

上面程序就是将前面的安全调用程序中的“?.”安全调用改为“!!.”强制调用，当可空变量b为null时，b!!.length将会引发NPE。与前面的安全调用类似的是，强制调用也可作用于let()函数，此时不管item元素是否为null，程序都会对该元素调用let()函数，因此也可能导致NPE。

2.9 字符串

Kotlin使用String代表字符串。字符串表示一个有序的字符集合，例如在前面代码中看到的"fkit.org"、"crazyit.org"等代表字符串，字符串也可被当成多个字符的有序集合。

2.9.1 字符串类型

String类型是一种快速、现代化的字符串实现，字符串中的每一个字符都由独立的Unicode字符组成，String允许通过形如s[i]的格式来访问字符串指定索引处的字符，也可通过for循环遍历字符串中的每一个字符。例如如下代码。

程序清单：codes\02\2.9\StringQs.kt

```
fun main(args: Array<String>) {
    var str = "fkjava.org"
    println(str[2]) // 输出 j
    println(str[3]) // 输出 a
    println(str[6]) // 输出.
    // 遍历 str 字符串中的每一个字符
    for (c in str) {
        println(c)
    }
}
```

上面程序中的第一行粗体字代码使用s[i]的形式根据索引访问字符串中指定位置的字符；第二行粗体字代码则使用for循环遍历字符串所包含的每一个字符。

Kotlin的字符串有两种字面值（Literal），分别如下。

- **转义字符串**：转义字符串可以有转义字符，转义字符串很像Java字符串。
- **原始字符串**：原始字符串可以包含换行和任意文本。原始字符串需要用3个引号引起来。

例如如下程序。

程序清单：codes\02\2.9\StringTest.kt

```
fun main(args: Array<String>) {
    // 定义普通字符串
    var str = "fkjava.org"
    println(str.length)
    // 定义原始字符串
    val txt = """
        |天上白玉京，
        |十二楼五城。
        |仙人抚我顶，
        |结发受长生。
    """.trimMargin()
    println(txt)
}
```

上面程序中的第一行粗体字代码使用的是转义字符串，这种字符串和普通的 Java 字符串基本相同；第二段粗体字代码使用的则是 Kotlin 原始字符串——用 3 个引号引起来的部分都是字符串内容，包括换行符等各种特殊字符。

编程时考虑到程序格式，往往会在原始字符串中进行一些缩进，但这些缩进并不是原始字符串希望包含的，此时即可使用 trimMargin()方法来去掉原始字符串前面的缩进——在默认情况下，Kotlin 使用竖线（|）作为边界符。也就是说，所有竖线（|）之前的内容都会被去掉。

如果有必要，开发者也可使用其他字符作为边界符，此时就需要在 trimMargin()方法中传入该边界符作为参数。例如如下程序（程序清单同上）：

```
    // 定义原始字符串
    val txt2 = """
        ^天上白玉京，
        ^十二楼五城。
        ^仙人抚我顶，
        ^结发受长生。
    """.trimMargin("^")
    println(txt2)
```

2.9.2 字符串模板

Kotlin 允许在字符串中嵌入变量或表达式，只要将变量或表达式放入${}中即可，这样 Kotlin 将会把该变量或表达式的值嵌入该字符串中。

字符串模板不仅可以在普通字符串中使用，也可以在原始字符串中使用。例如如下程序。

程序清单：codes\02\2.9\StringTemplate.kt

```
fun main(args: Array<String>) {
    val bookPrice = 79
    // 在字符串模板中嵌入变量
    var s = "图书价格是：${bookPrice}"
    println(s)
    val rand = java.util.Random() // 创建 Java 的 Random 对象
    // 在字符串模板中嵌入方法调用
    var s2 = "伪随机数是：${rand.nextInt(10)}"
    println(s2)
    val myStr = "fkjava.org"
    println("${myStr}的长度是${myStr.length}")
    var bookName = "疯狂 Kotlin 讲义"
    val poem = """
```

```
            |图书名是${bookName},
            |图书价格是${bookPrice},
            |随机促销价是${rand.nextInt(10) + 50}
        """.trimMargin()
        println(poem)
    }
```

上面程序中的第一行粗体字代码在字符串模板中嵌入了一个普通变量，第二行粗体字代码在字符串模板中嵌入了调用 Random 对象的 nextInt()方法；第三行粗体字代码则在字符串模板中嵌入了调用 String 对象的 length 属性；第四段粗体字代码在原始字符串中嵌入了变量、表达式，这也是允许的。

2.9.3 Kotlin 字符串的方法

Kotlin 的 String 与 Java 的 String 并不是同一个类，因此它们的方法略有不同，但是这两个类所提供的功能大致相似。实际上，Kotlin 的 String 类提供了更多的方法，如提供了一系列 toXxx()方法将字符串转换成数值。例如如下代码。

程序清单：codes\02\2.9\StringFunTest.kt

```
fun main(args: Array<String>) {
    val s1 = "2.34"
    // 将 s1 字符串转换成 Double 类型
    val d: Double = s1.toDouble()
    println(d)
    var s2 = "45"
    // 将 s2 字符串转换成 Int 类型
    var i: Int = s2.toInt()
    println(i)
}
```

此外，Kotlin 的 String 也提供了一些额外的工具方法，如将字符串首字母大写、小写的方法。例如如下代码（程序清单同上）：

```
    val str = "fkjava.org"
    println(str.capitalize()); // 首字母大写
    println(str.decapitalize()); // 首字母小写
```

以及返回两个字符串相同的前缀、后缀的方法。例如如下代码（程序清单同上）：

```
    var str2 = "crazyit.org"
    // 返回两个字符串相同的前缀
    println(str2.commonPrefixWith("crazyjava.org"));
    // 返回两个字符串相同的后缀
    println(str2.commonSuffixWith("fkit.org"));
```

还有 Java 的 String 的 contains()方法不支持使用正则表达式，但 Kotlin 的 String 的 contains()方法支持使用正则表达式匹配。例如如下代码（程序清单同上）：

```
    var str3 = "java886"
    // 判断 str3 是否包含 3 个连续的数字
    println(str3.contains(Regex("\\d{3}")))
```

关于 Kotlin 字符串还有不少非常方便的工具方法，具体可以参考 Kotlin 核心库的关于 String 的 API 文档：https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/index.html。

2.10 类型别名

Kotlin 提供了类似于 C 语言中的 typedef 的功能：可以为已有的类型指定另一个可读性更强的名字。Kotlin 提供了 typealias 来定义类型别名。typealias 语句的语法格式为：

```
typealias 类型别名 = 已有类型
```

如果类型名太长，我们可以使用较短的新名称来替代原类型名。这样有助于缩短较长的泛型类型。例如，通常缩短集合类型是很有吸引力的。

```
// 为Set<Network.Node>指定更短的别名：NodeSet
typealias NodeSet = Set<Network.Node>
// 为MutableMap<K, MutableList<File>>指定更短的别名：FileTable<K>
typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

接下来即可直接使用 NodeSet 和 FileTable<K>命名变量。

```
var set: NodeSet
var table: FileTable<String>
```

很多时候，我们也可以通过定义别名为内部类起一个更短的名字。例如如下程序。

程序清单：codes\02\2.10\AliasTest.kt

```
class A {
    inner class Inner
}
class B {
    inner class Inner
}
// 为A.Inner 内部类指定别名
typealias AInner = A.Inner
// 为B.Inner 内部类指定别名
typealias BInner = B.Inner
fun main(args: Array<String>) {
    // 使用 AInner 定义变量、调用对象
    var a: AInner = A().AInner()
    // 使用 BInner 定义变量、调用对象
    var b = B().BInner()
}
```

上面程序中的粗体字代码为 A.Inner、B.Inner 两个内部类分别指定了别名 AInner 和 BInner，接下来程序即可使用 AInner、BInner 别名类定义变量、创建对象。

此外，Kotlin 自己也大量利用了别名这个功能。比如 Kotlin 利用别名建立了 Kotlin 类和 Java 类之间的关系。如下代码是 Kotlin 集合体系中定义的别名。

```
typealias kotlin.ArrayList<E> = java.util.ArrayList<E>
typealias kotlin.HashMap<K, V> = java.util.HashMap<K, V>
typealias kotlin.HashSet<E> = java.util.HashSet<E>
typealias kotlin.LinkedHashMap<K, V> = java.util.LinkedHashMap<K, V>
typealias kotlin.LinkedHashSet<E> = java.util.LinkedHashSet<E>
typealias kotlin.RandomAccess = java.util.RandomAccess
```

与 Java 相比，Kotlin 的 Lambda 表达式是一个差别较大的特色功能：Java 的 Lambda 表达式的类型是函数式接口，而 Kotlin 的 Lambda 表达式的类型直接就是函数类型，因此 Kotlin 也允许为 Lambda 表达式的类型指定别名。例如如下代码。

程序清单：codes\02\2.10\AliasTest.kt

```
// 为(T) -> Boolean 类型指定别名 Predicate<T>
typealias Predicate<T> = (T) -> Boolean
```

```
// 使用 Predicate<String>定义变量，该变量的值是一个 Lambda 表达式
val p: Predicate<String> = { it.length > 4 }
// 为 filter()方法传入 p 参数，只保留长度大于 4 的字符串
println(arrayOf("Java", "Objective-C", "Go", "Kotlin").filter(p))
```

2.11 本章小结

本章介绍的是 Kotlin 编程的最基础内容。本章内容围绕 Kotlin 变量和常量展开，详细介绍了定义变量和常量的语法知识。由于 Kotlin 是强类型的语言，因此定义变量和常量时必须显式或隐式指定数据类型。本章的重点是掌握 Kotlin 提供的各种基础数据类型，如整型、浮点型、布尔型和字符串类型，并掌握整型和浮点型之间进行类型转换的语法及注意点。

除此之外，Kotlin 的可空和非空类型也是值得注意的知识点。Kotlin 的普通类型都不能接受 null 值，只有可空类型才能接受 null 值。使用可空类型时，需要注意可空类型的空值判断，或使用安全调用，这样就可以避免程序出现 NPE。

CHAPTER

3

第3章 运算符和表达式

本章要点

- 单目前缀运算符及对应的方法
- 自加和自减运算符及对应的方法
- 双目算术运算符及对应的方法
- in 和!in 及对应的方法
- 索引访问运算符及对应的方法
- 调用运算符及对应的方法
- 广义赋值运算符及对应的方法
- 相等和不等运算符及对应的方法
- 比较运算符及对应的方法
- 位运算符
- 区间运算符
- 半开区间运算符
- 反向区间和区间步长
- 运算符重载

学习完前一章介绍的各种类型的变量和常量之后，读者必然需要对这些变量和常量进行必要的运算和处理，这就需要使用 Kotlin 的各种运算符，通过运算符可以把变量、常量连接成表达式。

Kotlin 语言提供了一系列功能丰富的运算符，这些运算符包括所有的算术运算符，还包括比较运算符、逻辑运算符、区间运算符和位运算符等。Kotlin 基本支持 Java 的全部运算符（可能有些在写法上存在差异），Kotlin 不支持三目运算符（因为 Kotlin 可用 if 表达式代替三目运算符）。

需要指出的是，Kotlin 的很多运算符其实等价于 operator 修饰的、特定函数名的函数，因此 Kotlin 的运算符非常强大，其不仅能作用于普通数值类型的变量和常量，也可作用于任何自定义的类——只要它们提供 operator 修饰的、特定函数名的函数即可。本章也会详细介绍 Kotlin 运算符的这种强大功能。

3.1 与 Java 相同的运算符

Java 支持的运算符包括算术运算符、赋值运算符、扩展后的赋值运算符、比较运算符、逻辑运算符，Kotlin 也完全支持这些运算符。

Kotlin 不支持 Java 的三目运算符——Kotlin 使用 if 表达式代替了三目运算符。

Kotlin 的位运算符与 Java 的位运算符也略有区别。

但有一点需要说明：Kotlin 的运算符都是以方法形式来实现的，这些运算符都具有特定的符号（如“+”或“*”）和固定的优先级。

各种运算符对应的方法名都是固定的，我们只要为某类型提供了特定名称（比如双目＋运算符对应的方法名为 plus）的方法，成员方法或扩展方法都可，接下来即可对该类型的对象使用“+”进行运算——因此，Kotlin 的所有运算符的功能都是广义的，不仅能作用于数值型、字符串，而且也可作用于任意自定义的 Kotlin 类。本章后面内容会介绍运算符重载的知识。

3.1.1 单目前缀运算符

单目前缀运算符有+、-、!这三个。它们对应的固定方法如表 3.1 所示。

表 3.1 单目前缀运算符及对应的方法

运 算 符	对应的方法
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

例如如下程序。

程序清单：codes\03\3.1\UnaryTest.kt

```
fun main(args: Array<String>) {
    var a = 20;
    // 使用运算符
    val b = -a;
    // 调用方法
    val c = a.unaryMinus();
    println("b: ${b}, c: ${c}");
    val flag = true
    // 使用运算符
    val notFlag1 = !flag
```

```
    // 调用方法
    val notFlag2 = flag.not()
    println("notFlag1: ${notFlag1}, notFlag2: ${notFlag2}");
}
```

从上面的粗体字代码可以看出，-a 与 a.unaryMinus()的效果是完全一样的；!flag 与 flag.not()的效果也是完全一样的。因此，以后读者在查阅 API 时发现某个类有 unaryPlus()、unaryMinus()、not()方法，那就说明可对该类的实例使用单目前缀运算符+、-、!进行运算。

3.1.2 自加和自减运算符

自加（++）和自减（--）运算符及对应的固定方法如表 3.2 所示。

表 3.2 自加和自减运算符及对应的方法

运算符	对应的方法
a++	a.inc()
a--	a.dec()

由于++、--放在变量前后是有区别的，因此自加和自减运算符对应的 inc()和 dec()两个方法还不完全等同于 a++、a--。

当++、--放在变量前面时，执行过程如下：

1. 先对变量调用 inc()、dec()方法，并将方法返回值赋给变量。
2. 自加或自减表达式返回变量的新值。

当++、--放在变量后面时，执行过程如下：

1. 先用一个临时变量缓存变量的值。
2. 对变量调用 inc()、dec()方法，并将方法返回值赋给变量。
3. 自加或自减表达式返回临时变量的值。

例如如下程序。

程序清单：codes\03\3.1\SelfInc.kt

```
fun main(args: Array<String>) {
    var a = 20
    a++
    println("a: ${a}")  // 输出 21
    // 以下代码大致相当于 a++
    a = a.inc()
    println("a: ${a}")  // 输出 22
}
```

由此可见，以后读者在查阅 API 时发现某个类有 inc()、dec()方法，那就说明可对该类的实例使用++、--进行运算。

3.1.3 双目算术运算符

双目算术运算符及对应的固定方法如表 3.3 所示。

表 3.3 双目算术运算符及对应的方法

运 算 符	对应的方法
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)

续表

运 算 符	对应的方法
a % b	a.rem(b)
a..b	a.rangeTo(b)

例如如下程序。

程序清单：codes\03\3.1\Arthimetic.kt

```
fun main(args: Array<String>) {
    // 加法运算
    println(5 + 6)
    println(5.plus(6))
    // 乘法运算
    println(2.3 * 3.4)
    println(2.3.times(3.4))
    // 求余运算
    println(2.5 % 1.2)
    println(2.5.rem(1.2))
}
```

由此可见，以后读者在查阅 API 时发现某个类有带一个参数的 plus()、minus()、times()、div()、rem()和 rangeTo()方法，那就说明可对该类的实例使用以上双目算术运算符。



提示：

此处暂时不介绍 rangeTo()方法和“..”运算符，本章中的区间运算符就是专门介绍它们的。

3.1.4 in 和!in 运算符

in 和!in 运算符及对应的固定方法如表 3.4 所示。

表 3.4 in 和!in 运算符及对应的方法

运 算 符	对应的方法
a in b	b.contains(a)
a !in b	!b.minus(a)

例如如下程序。

程序清单：codes\03\3.1\InTest.kt

```
fun main(args: Array<String>) {
    var str = "fkjava.org"
    // 调用 String 的 contains 方法判断
    println( str.contains("java") )
    // 使用 in 运算符判断
    println("java" in str)
    val array = arrayOf(24, 45, 100, -3, 30)
    // 调用 Array 的 contains 方法判断
    println(array.contains(100))  // 输出 true
    // 使用 in 运算符判断
    println(100 in array)  // 输出 true
}
```

从上面代码可以看出，由于 String 类中存在带一个参数的 contains()方法，因此在 Kotlin 中即可使用 in 运算符进行计算。

由此可见，以后读者在查阅 API 时发现某个类有带一个参数的 contains()方法，那就说明可对该类的实例使用 in、!in 运算符。



提示：

这就是 Kotlin 比 Java 简洁的地方。Java 程序要判断某个 String 是否包含某个子串，只能调用 contains()方法进行判断，但 Kotlin 则只要使用 in 运算符即可搞定。Kotlin 提供了大量这种“语法糖”来让开发者享受编程。

3.1.5 索引访问运算符

索引访问运算符及对应的固定方法如表 3.5 所示。

表 3.5 索引访问运算符及对应的方法

运 算 符	对应的方法
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1,..., i_n]	a.get(i_1,..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1,..., i_n] = b	a.set(i_1,..., i_n, b)

例如如下程序。

程序清单：codes\03\3.1\IndexTest.kt

```
fun main(args: Array<String>) {
    var str = "fkjava.org"
    // 根据 get 方法获取指定索引处的字符
    println(str.get(2))
    // 使用索引访问运算符来获取指定索引处的字符
    println(str[2])
    // 创建 Java 的 ArrayList 集合
    var list = java.util.ArrayList<String>()
    list.add("Java")
    list.add("Kotlin")
    list.add("Go")
    // 使用索引访问运算符来获取指定索引处的 List 集合元素
    println(list[1])  // 输出 Kotlin
    // 使用索引访问运算符来修改指定索引处的 List 集合元素
    list[2] = "Swift"
    println(list)
}
```

从上面的粗体字代码可以看出，由于 String 类提供了 get(index)方法，因此程序即可通过 str[2]这样的索引访问运算符来获取指定索引处的字符；而 Java 的 ArrayList 则提供了 get(index)和 set(index, val)方法，因此程序可通过索引访问运算符来获取或修改指定索引处的集合元素。



提示：

这其实也是 Kotlin 的“语法糖”。

3.1.6 调用运算符

调用运算符及对应的固定方法如表 3.6 所示。

表 3.6 调用运算符及对应的方法

运 算 符	对应的方法
a()	a.invoke()
a(b)	a.invoke(b)
a(b1, b2)	a.invoke(b1, b2)
a(b1, b2, b3, ...)	a.invoke(b1, b2, b3, ...)

从表 3.6 可以看出，调用运算符其实就是省略了 invoke 方法名。例如如下程序。

程序清单：codes\03\3.1\InvokeTest.kt

```
fun main(args: Array<String>) {
    val s = "java.lang.String"
    // 使用反射获取 String 类的 length()方法
    val mtd = Class.forName(s).getMethod("length")
    // 使用传统方法，使用 Method 对象的 invoke()方法
    println(mtd.invoke("java"))  // 输出 4
    // 使用调用运算符
    println(mtd("java")  // 输出 4
}
```

上面两行粗体字代码的本质是一样的。由于 Method 类提供了 invoke()方法，因此程序既可按传统 Java 方式调用 invoke()方法，也可直接使用调用运算符进行调用。

3.1.7 广义赋值运算符

广义赋值运算符及对应的固定方法如表 3.7 所示。

表 3.7 广义赋值运算符及对应的方法

运 算 符	对应的方法
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b)

这种广义的赋值运算符有些特殊，比如 a += b，实际上相当于 a = a + b，因此在程序中进行 a += b 运算时，往往并不需要 a 有 plusAssign()方法。

对于广义赋值操作，例如 a += b，编译器会先判断 plusAssign()方法是否存在，如果存在，则按如下步骤执行。

1. 如果 plus()方法也存在，Kotlin 将会报告错误（调用的目标方法不确定）。
2. 确保 plusAssign()没有返回值，否则报告错误。

注意：

Kotlin 的赋值运算不是表达式，因此不需要返回值。



3. 如果能通过前两步的检查，则转换为执行 a.plusAssign(b)。

如果 plusAssign()方法不存在，那么 a += b 将转换为 a = a + b 代码。

3.1.8 相等与不等运算符

相等（==）和不等（!=）运算符及对应的固定方法如表 3.8 所示。

表 3.8 相等和不等运算符及对应的方法

运 算 符	对应的方法
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null)))

从表 3.8 可以看出，Kotlin 的“==”不再比较两个变量是否引用同一个对象。实际上，“==”与 equals()比较基本是等义的，只不过“==”比较是空指针安全的。例如如下程序。

程序清单：codes\03\3.1\EqualsTest.kt

```
fun main(args: Array<String>) {
    var s1 = java.lang.String("java")
    var s2 = java.lang.String("java")
    println(s1 == s2);  // 输出 true
    println(s1.equals(s2)); // 输出 true
}
```

上面两行粗体字代码的本质是一样的。因此，虽然 s1、s2 两个变量指向两个不同的字符串，但这两行粗体字代码依然会输出 true。

Java 提供的“==”和“!=”在 Kotlin 中则由“===”和“!==”代替了，如下代码将会输出 false。

```
println(s1 === s2); // 输出 false
```

上面进行 s1 === s2 判断时才要求 s1、s2 两个变量指向同一个对象。

3.1.9 比较运算符

比较运算符及对应的固定方法如表 3.9 所示。

表 3.9 比较运算符及对应的方法

运 算 符	对应的方法
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

从表 3.9 可以看出，比较运算符其实就是由 compareTo()方法来实现的，而该方法是 Comparable 接口中定义的方法，因此原来 Java 类中支持使用 compareTo()方法比较大小的对象，都可使用比较运算符进行计算。例如如下代码。

程序清单：codes\03\3.1\CompareTest.kt

```
fun main(args: Array<String>) {
    var s1 = "java"
    var s2 = "kotlin"
    // 下面两行代码是相同的
    println(s1 > s2) // 输出 false
    println(s1.compareTo(s2) > 0) // 输出 false
    var date1 = java.util.Date()
    var date2 = java.util.Date(System.currentTimeMillis() - 1000)
    println(date1 > date2) // 输出 true
    println(date1.compareTo(date2) > 0) // 输出 true
}
```

上面程序中使用比较运算符对 String、Date 两个类的实例进行了大小比较，由于这两个类都实现了 Comparable 接口，因此完全可以使用比较运算符来比较大小（至于各种数值类型，

就更可以比较大小了），Kotlin 将会把大小比较运算符转换为通过 compareTo()方法进行比较。



提示：

String 比较大小的规则是按字符的编号大小进行比较的——先比较两个字符串的首字母；如果首字母相同，再比较第二个字母……依此类推，这其实是 Java 的知识。

由此可见，使用 Kotlin 编程时，只要某个类实现了 Comparable 接口，那么该类的实例即可使用比较运算符来比较大小。

3.2 位运算符

虽然 Kotlin 也提供了与 Java 功能完全相同的位运算符，但这些位运算符都不是以特殊字符给出的，而是以 infix 函数的形式给出的，因此程序只能用函数名来执行这些位运算符。

Kotlin 支持的位运算符同样有如下 7 个。

- and(bits)：按位与。当两位同时为 1 时才返回 1。
- or(bits)：按位或。只要有一位为 1，即可返回 1。
- inv(bits)：按位非。单目运算符，将操作数的每个位（包括符号位）全部取反。
- xor(bits)：按位异或。当两位相同时返回 0，不同时返回 1。
- shl(bits：左移运算符。
- shr(bits)：右移运算符。
- ushr(bits)：无符号右移运算符。

Kotlin 的位运算符只能对 Int 和 Long 两种数据类型起作用。位运算符的运算法则如表 3.10 所示。

表 3.10 位运算符的运算法则

第一个运算数	第二个运算数	按位与	按位或	按位异或
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

按位非只需要一个操作数，这个运算符将把操作数在计算机底层的二进制码按位（包括符号位）取反。如下代码测试了按位与和按位或运算的运行结果。

程序清单：codes\03\3.2\BitOperatorTest.kt

```
println(5 and 9) // 将输出 1
println(5 or 9) // 将输出 13
```

程序执行的结果是：5 and 9 的结果是 1，5 or 9 的结果是 13。下面介绍其运算原理。

5 的二进制码是 00000101（省略了前面的 24 个 0），而 9 的二进制码是 00001001（省略了前面的 24 个 0）。运算过程如图 3.1 所示。

```
  00000101      00000101
and 00001001  or 00001001
----------    ----------
  00000001      00001101
```

图 3.1 按位与和按位或运算过程

下面是按位异或和按位取反的执行代码（程序清单同上）。

```
println((-5).inv()) // 将输出 4
println(5 xor 9) // 将输出 12
```

程序执行-5 按位取反的结果是 4，执行 5 xor 9 的结果是 12。下面介绍其运算原理。

-5 按位取反的运算过程如图 3.2 所示。

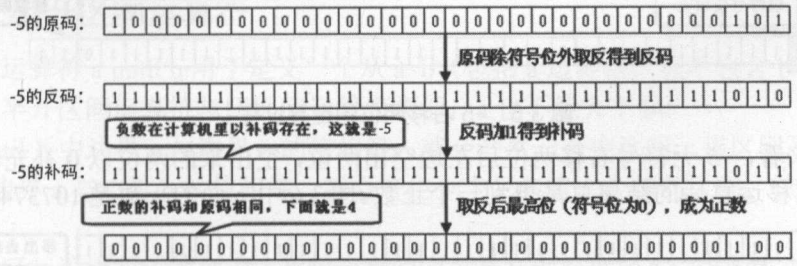


图 3.2 -5 按位取反的运算过程

5 xor 9 的运算过程如图 3.3 所示。

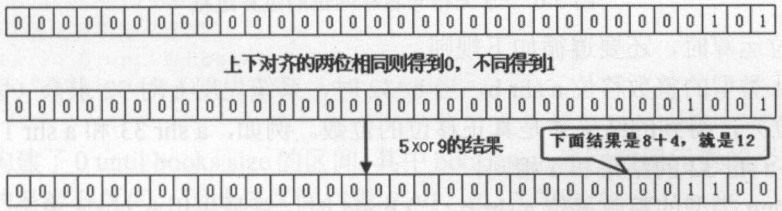


图 3.3 5 xor 9 的运算过程

左移运算符是将操作数的二进制码整体左移指定位数，左移后右边空出来的位以 0 填充。例如如下代码（程序清单同上）：

```
println(5 shl 2) // 输出 20
println(-5 shl 2) // 输出-20
```

下面以-5 为例来介绍左移运算符的运算过程，如图 3.4 所示。

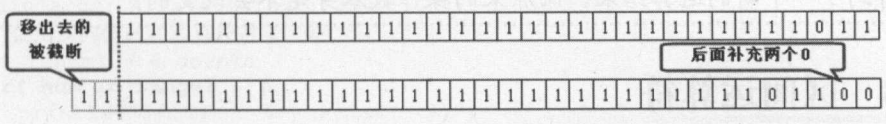


图 3.4 -5 左移两位的运算过程

在图 3.4 中，上面的 32 位数是-5 的补码，左移两位后得到一个二进制补码，这个二进制补码的最高位是 1，表明是一个负数，换算成十进制数就是-20。

Kotlin 的右移运算符有两个：shr 和 ushr。对于 shr 运算符而言，把第一个操作数的二进制码右移指定位数后，左边空出来的位以原来的符号位填充，即如果第一个操作数原来是正数，则左边补 0；如果第一个操作数是负数，则左边补 1。ushr 是无符号右移运算符，它把第一个操作数的二进制码右移指定位数后，左边空出来的位总是以 0 填充。

看下面代码（程序清单同上）：

```
// 输出-2
println(-5 shr 2)
// 输出 1073741822
println(-5 ushr 2)
```

下面用示意图来说明 shr 和 ushr 运算符的运算过程。

从图 3.5 来看，-5 右移两位后左边空出两位，空出来的两位以符号位补充。从图中可以看出，右移运算后得到的结果的正负与第一个操作数的正负相同。右移后的结果依然是一个负数，这是一个二进制补码，换算成十进制数就是-2。

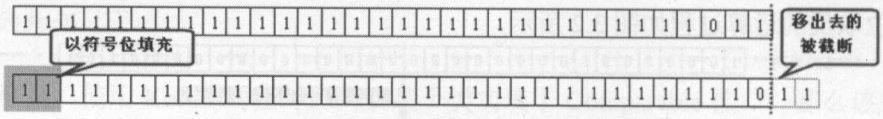


图 3.5 -5 右移两位的运算过程

从图 3.6 来看，-5 无符号右移两位后左边空出两位，空出来的两位以 0 补充。从图中可以看出，无符号右移运算后的结果总是得到一个正数。图 3.6 中下面的正数是 1073741822(2^{30}-2)。

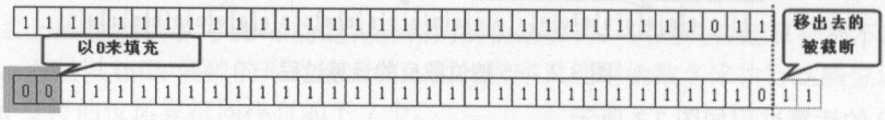


图 3.6 -5 无符号右移两位的运算过程

在进行移位运算时，还要遵循如下规则。

- 对于 Int 类型的整数移位 a shr b，当 b>32 时，系统先用 b 对 32 求余（因为 Int 类型只有 32 位），得到的结果才是真正移位的位数。例如，a shr 33 和 a shr 1 的结果完全一样，而 a shr 32 的结果和 a 相同。
- 对于 Long 类型的整数移位 a shr b，当 b>64 时，总是先用 b 对 64 求余（因为 Long 类型是 64 位），得到的结果才是真正移位的位数。

注意：

当进行移位运算时，只要被移位的二进制码没有发生有效位的数字丢失（对于正数而言，通常指被移出的位全部都是 0），不难发现左移 *n* 位就相当于乘以 2 的 *n* 次方，右移 *n* 位则相当于除以 2 的 *n* 次方。不仅如此，在进行移位运算时，只是得到了一个新的运算结果，而原来的操作数本身是不会改变的。



3.3 区间运算符

Kotlin 提供了两个区间运算符，即闭区间运算符和半开区间运算符，它们都可以非常方便地构建一种数据结构，这种数据结构可包含特定区间内的所有值。

3.3.1 闭区间运算符

闭区间运算符 a .. b 用于定义一个从 a~b（包括 a、b 边界值）的所有值的区间。对于闭区间运算符而言，a 不能大于 b，否则程序运行时将会报错。

Kotlin 的 for-in 循环可用于遍历区间内的所有数据。例如如下代码。

程序清单：codes\03\3.3\RangeTest.kt

```
fun main(args: Array<String>) {
    // 使用闭区间运算符定义区间
    var range1 = 2 .. 6
    for( num in range1 ) {
```

```
        println("${num} * 5 = ${num * 5}")
    }
}
```

上面粗体字代码构建了一个区间，该区间包含2~6的所有数值，就是2、3、4、5、6这5个值。

3.3.2 半开区间运算符

半开区间运算符a until b用于定义一个从a~b（包括a边界值，但不包含b边界值）的所有值的区间。半开区间运算符与闭区间运算符类似，a也不能大于b。

如果a until b中边界a与边界b的值相等，就会产生一个空区间，该区间不包含任何值；如果a .. b中边界a与边界b的值相等，就会产生一个只包含一个值的区间，该区间只包含一个边界值。

利用半开区间遍历数组等列表（数组元素的索引是0到长度减1）时非常方便，例如如下程序（程序清单同上）：

```
// 定义数组
val books = arrayOf("Swift" , "Kotlin" , "C" , "C++")
// 使用半开区间运算符定义区间
for ( index in 0 until books.size) {
    println("第${index+1}种语言是：${books[index]}")
}
```

上面程序构建了0 until books.size的区间，其中books.size返回数组元素的个数，也就是4。而且此处使用的是半开区间，因此该区间代表0、1、2、3等数值。

3.3.3 反向区间

如果程序希望区间可以从大到小，则可使用downTo运算符（其实是一个infix函数），该运算符同样构建一个闭区间。对于a downTo b而言，此时要求b不能大于a。

例如，如下程序使用downTo定义了反向区间。

程序清单：codes\03\3.3\DownToTest.kt

```
fun main(args: Array<String>) {
    // 使用反向闭区间运算符定义区间
    var range1 = 6 downTo 2
    for( num in range1 ) {
        println("${num} * 5 = ${num * 5}")
    }
}
```

上面程序使用downTo定义的区间将会从6逆序降到2。上面程序的输出结果如下：

```
6 * 5 = 30
5 * 5 = 25
4 * 5 = 20
3 * 5 = 15
2 * 5 = 10
```

3.3.4 区间步长

前面我们见到的所有区间的默认步长都是1，也就是区间内两个值之间的差值是1。而通过step运算符（其实是一个infix函数）可以显式指定区间的步长。例如如下程序（程序清单同上）：

```
// 为反向闭区间指定步长
```

```
for( num in 7 downTo 1 step 2 ) {
    println("${num} * 5 = ${num * 5}")
}
```

上面程序使用 downTo 定义的区间将会从 7 逆序降到 1，且步长为 2。上面程序的输出结果如下：

```
7 * 5 = 35
5 * 5 = 25
3 * 5 = 15
1 * 5 = 5
```

3.4 运算符重载

正如 3.1 节所介绍的，Kotlin 的运算符都是靠特定名称的方法支撑的，因此只要重载这些名称的方法，我们就可以为任意类添加这些运算符。

重载运算符的方法需要用 operator 修饰符进行标记。

3.4.1 重载单目前缀运算符

表 3.1 已经给出了单目前缀运算符对应的方法名，因此只要为任意类定义名为 unaryPlus()、unaryMinus()、not()，且以 operator 修饰的方法，程序即可对该类的实例使用+、-、!单目前缀运算符。例如如下程序。

程序清单：codes\03\3.4\UnaryTest.kt

```
data class Data(val x: Int, val y: Int){
    // 为 Data 类定义一个 unaryMinus()方法
    operator fun unaryMinus(): Data {
        return Data(-x, -y)
    }
}
// 以扩展方法的形式为 Data 类定义 not()方法
operator fun Data.not(): Data {
    return Data(-x, -y)
}
fun main(args: Array<String>) {
    val d = Data(4, 10)
    println(-d) // 输出 Data(x=-4, y=-10)
    println(!d) // 输出 Data(x=-4, y=-10)
}
```

上面程序中定义了一个非常简单的 Data 类，并在该类中定义了一个 operator 修饰的 unaryMinus()方法，该方法返回一个将 Data 的 x、y 两个属性取反的 Data 实例，这意味着我们可以对 Data 实例应用“-”这个单目前缀运算符。



提示：

关于 Kotlin 定义类、扩展方法的知识，请参考本书 8.1 节内容。

此外，上面程序还以扩展方法的形式为 Data 类定义了一个 not()方法，该方法同样返回一个将 Data 的 x、y 两个属性取反的 Data 实例，这意味着我们可以对 Data 实例应用“!”这个单目前缀运算符。

上面程序中 main()方法的最后两行对 Data 对象应用了“-”和“!”这两个运算符，这两个表达式的输出结果是一样的。

3.4.2 重载自加和自减运算符

表 3.2 已经给出了自加和自减运算符对应的方法名，因此只要为任意类定义名为 inc()、dec()，且以 operator 修饰的方法，程序即可对该类的实例使用++、--运算符。例如如下程序。

程序清单：codes\03\3.4\SelfInc.kt

```
data class Data(val x: Int, val y: Int){
    // 为 Data 类定义一个 inc()方法
    operator fun inc(): Data {
        return Data(x + 1, y + 1)
    }
}
// 以扩展方法的形式为 Data 类定义 dec()方法
operator fun Data.dec(): Data {
    return Data(x - 1, y - 1)
}
fun main(args: Array<String>) {
    var d = Data(4, 10)
    println(d++) // 先用，再自加，输出 Data(x=4, y=10)
    println(d) // 输出自加后的 d：Data(x=5, y=11)
    var dd = Data(9, 20)
    println(--dd) // 先自减，再用，输出 Data(x=8, y=19)
    println(dd) // 输出自加后的 dd：Data(x=8, y=19)
}
```

上面程序以普通方法和扩展方法的形式为 Data 类定义了 inc()和 dec()两个方法，并使用 operator 修饰了这两个方法，因此接下来程序即可对 Data 实例使用自加和自减运算符执行运算。

当程序对 Data 对象使用自加运算符时，该对象的 x、y 两个属性都会加 1；当程序对 Data 对象使用自减运算符时，该对象的 x、y 两个属性都会减 1。至于++、--放在变量前面和后面的差异，其实与 Java 语言中++、--放在变量前面和后面的差异完全一样，此处不再深入展开介绍。

3.4.3 重载双目算术运算符

表 3.3 已经给出了双目算术运算符对应的方法名，因此只要为任意类定义名为 plus()、minus()、times()、div()、rem()、rangeTo()，且以 operator 修饰的方法，程序即可对该类的实例使用+、-、*、/、%、..这些运算符。例如如下程序。

程序清单：codes\03\3.4\ArthimethicTestKt.class.kt

```
data class Point(val x: Int, val y: Int){
    // 为 Point 类定义一个 minus()方法
    operator fun minus(target: Point): Double{
        return Math.hypot((this.x - target.x).toDouble()
            , (this.y - target.y).toDouble())
    }
}
// 以扩展方法的形式为 Point 类定义 times()方法
operator fun Point.times(target: Point): Int {
    return Math.abs(this.x - target.x) *
        Math.abs(this.y - target.y)
}
fun main(args: Array<String>) {
    var p1 = Point(4, 10)
    var p2 = Point(5, 15)
    var distance = p1 - p2;
    println("p1 与 p2 的距离为：${distance}")
    var area = p1 * p2
```

```
    println("p1 与 p2 围成矩形的面积为：${area}")
}
```

上面程序中定义了一个 Point 类，接下来程序在 Point 类中定义了一个 minus()方法，该方法用于计算两个 Point 之间的距离；此外，程序还通过扩展方法的形式为 Point 类定义了一个 times()方法，该方法用于计算两个 Point 所围成矩形的面积。

只要为 Point 类定义了 minus()、times()两个方法，接下来程序即可用-、*两个运算符来计算两个运算符之间的距离，以及所围成矩形的面积，如上面程序所示。

3.5 本章小结

本章的知识点以表达式为中心，程序使用运算符将已有的值、变量和常量连接起来，这样就可以形成表达式，因此学习本章的重点就是掌握 Kotlin 提供的各种运算符，包括单目前缀运算符、自加和自减运算符、双目算术运算符、in 和!in 运算符、索引访问运算符、调用运算符、广义赋值运算符、比较运算符、位运算符、区间运算符。此外，由于 Kotlin 的运算符与 operator 修饰的、特定函数名的函数是等价的，因此读者还需要掌握与各运算符等价的函数，并掌握重载运算符的方法。

CHAPTER

4

第4章 流程控制

本章要点

- 顺序结构
- if 分支的功能和用法
- when 分支的功能和用法
- when 分支处理范围
- when 条件分支
- while 循环
- do while 循环
- for-in 循环
- 嵌套循环
- 控制循环结构

与 Java 类似，Kotlin 同样提供了两种基本的流程控制结构：分支结构和循环结构。其中分支结构用于实现根据条件来选择性地执行某段代码；循环结构则用于实现根据循环条件重复执行某段代码。Kotlin 同样提供了这两种流程控制结构的语法，Kotlin 提供了 if 和 when 两种分支语句，Kotlin 的 when 语句可以代替 Java 的 switch 语句，而且功能更强大；Kotlin 提供了 while、do while、for-in 循环，抛弃了 Java 原有的普通 for 循环。Kotlin 也提供了 break 和 continue 来控制程序的循环结构。

4.1 顺序结构

在任何编程语言中最常见的程序结构就是顺序结构。顺序结构就是程序从上到下一行行地执行，中间没有任何判断和跳转。

如果 Kotlin 程序的多行代码之间没有任何流程控制，则程序总是从上向下依次执行，排在前面的代码先执行，排在后面的代码后执行。这意味着：如果没有流程控制，Kotlin 程序的语句是一个顺序执行流，从上向下依次执行每条语句。

4.2 分支结构

Kotlin 提供了两种常见的分支控制结构：if 分支和 when 分支。其中 if 分支使用布尔表达式或布尔值作为分支条件来进行分支控制；而 when 分支则更适用于复杂的条件。

一般来说，当条件较为简单且可能的情况很少时，使用 if 分支；当条件比较复杂且可能的情况很多时，可以考虑使用 when 分支。

4.2.1 if 分支

Kotlin 的 if 分支既可作为语句使用，也可作为表达式使用。下面先介绍 if 分支作为语句使用的情形。if 语句可使用任意表达式作为分支条件来进行分支控制。与 Java 相似，Kotlin 的 if 语句有如下三种形式。

第一种形式：

```
if (expression) {
    statements...
}
```

第二种形式：

```
if (expression) {
    statements...
}
else {
    statements...
}
```

第三种形式：

```
if (expression) {
    statements...
}
else if (expression) {
    statements...
}
...// 可以有零个或多个 else if 语句
```

```
else {// 最后的 else 语句也可以省略
    statement...
}
```

在上面 if 语句的三种形式中，第二种形式和第三种形式是相通的，如果第三种形式中的 else if 块不出现，则变成了第二种形式。

在上面的条件语句中，if (expression)、else if (expression)以及 else 后花括号括起来的多行代码被称为代码块，一个代码块通常被当成一个整体来执行（除非在运行过程中遇到 return、break、continue 等关键字），因此这个代码块也被称为条件执行体。例如如下程序。

程序清单：codes\04\4.2\IfTest.kt

```
fun main(args: Array<String>) {
    var age = 30
    if (age > 20) {
    // 只有当 age > 20 时，下面用花括号括起来的代码块才会执行
    // 用花括号括起来的语句是一个整体，要么一起执行，要么一起不执行
        println("年龄已经大于 20 岁了")
        println("20 岁以上的人应该学会承担责任...")
    }
}
```

如果 if(logic expression)、else if(logic expression)和 else 后的代码块只有一行语句，则可以省略花括号，因为单行语句本身就是一个整体，无须用花括号把它们定义成一个整体。下面代码完全可以正常执行（程序清单同上）。

```
// 定义变量 a，并为其赋值
val a = 5
if (a > 4)
    // 如果 a>4，则执行下面的执行体，只有一行代码作为代码块
    println("a 大于 4")
else
    // 否则，执行下面的执行体，只有一行代码作为代码块
    println("a 不大于 4")
```

通常建议不要省略 if、else、else if 后执行体的花括号，即使条件执行体只有一行代码，也要保留花括号，这样会有更好的可读性，而且保留花括号会减少发生错误的可能。例如如下代码，则不能正常执行（程序清单同上）。

```
// 定义变量 b，并为其赋值
var b = 5
if (b > 4)
    // 如果 b>4，则执行下面的执行体，只有一行代码作为代码块
    println("b 大于 4")
else
    // 否则，执行下面的执行体，只有一行代码作为代码块
    b--
    // 对于下面代码而言，它已经不再是条件执行体的一部分，因此总会执行
    println("b 不大于 4")
```

上面代码中以粗体字标识的代码行：println("b 不大于 4")，总会执行，因为这行代码并不属于 else 后的条件执行体，else 后的条件执行体就是 b--这一行代码。

注意：

if、else、else if 后的条件执行体要么是一个用花括号括起来的代码块，则这个代码块整体作为条件执行体；要么是单独的一条语句，甚至可能是一条空语句



（空语句是一个分号），那么就只有这条语句作为条件执行体。如果省略了 if 条件后条件执行体的花括号，那么 if 条件只控制紧跟该条件语句的一条语句。

如果 if 后有多条语句作为条件执行体，若省略了这个条件执行体的花括号，则会引起编译错误。看下面代码（程序清单同上）：

```
// 定义变量 c，并为其赋值
var c = 5
if (c > 4)
    // 如果 c>4，则执行下面的执行体，将只有 c--一行代码为执行体
    c--
    // 下面是一行普通代码，不属于执行体
    println("c 大于 4")
// 此处的 else 将没有 if 语句，因此编译出错
else
    // 否则，执行下面的执行体，只有一行代码作为代码块
    println("c 不大于 4")
```

在上面代码中，因为 if 后的条件执行体省略了花括号，则系统只把 c--一行代码作为条件执行体，当 c--语句执行结束后，if 语句也就执行结束了。后面的 println("c 大于 4")已经是一行普通代码了，不再属于条件执行体，从而导致 else 语句没有 if 语句，从而引起编译错误。

对于 if 分支，还有一个很容易出现的逻辑错误，这个逻辑错误并不属于语法问题，但引起错误的可能性更大。看下面程序。

程序清单：codes\04\4.2\IfErrorTest.kt

```
fun main(args: Array<String>) {
    var age = 45
    if (age > 20){
        println("青年人")
    }
    else if (age > 40) {
        println("中年人")
    }
    else if (age > 60) {
        println("老年人")
    }
}
```

表面上看，上面的程序没有任何问题：人的年龄大于 20 岁是青年人，年龄大于 40 岁是中年人，年龄大于 60 岁是老年人。但运行上面程序，就会发现打印结果是：青年人。而实际上希望 45 岁应判断为中年人——这显然出现了一个问题。

对于任何的 if else 语句，表面上看，else 后没有任何条件，或者 else if 后只有一个条件——但这不是真相：因为 else 的含义是“否则”——else 本身就是一个条件！这也是把 if、else 后代码块统称为条件执行体的原因，else 的隐含条件是对前面条件取反。因此，上面代码实际上可改写为如下形式。

程序清单：codes\04\4.2\IfErrorTest2.kt

```
fun main(args: Array<String>) {
    var age = 45
    if (age > 20) {
        println("青年人")
    }
    // 在原本的 if 条件中增加了 else 的隐含条件
    if (age > 40 && !(age > 20)) {
        println("中年人")
```

```
    }
    // 在原本的 if 条件中增加了 else 的隐含条件
    if (age > 60 && !(age > 20) && !(age > 40 && !(age > 20))) {
        println("老年人")
    }
}
```

此时就比较容易看出为什么发生上面的错误了。对于 age > 40 && !(age > 20)这个条件，又可改写成 age > 40 && age <= 20，这样永远也不会发生上面的错误了。对于 age > 60 && !(age > 20) && !(age > 40 && !(age > 20))这个条件，则更不可能发生错误了。因此，程序永远都不会判断中年人和老年人的情形。

为了达到正确的目的，可以把程序改写为如下形式。

程序清单：codes\04\4.2\IfCorrectTest.kt

```
fun main(args: Array<String>) {
    var age = 45
    if (age > 60) {
        println("老年人")
    }
    else if (age > 40) {
        println("中年人")
    }
    else if (age > 20) {
        println("青年人")
    }
}
```

运行程序，得到了正确结果。实际上，上面程序等同于下面代码。

程序清单：codes\04\4.2\IfCorrectTest2.kt

```
fun main(args: Array<String>) {
    var age = 45
    if (age > 60) {
        println("老年人")
    }
    // 在原本的 if 条件中增加了 else 的隐含条件
    if (age > 40 && !(age >60)){
        println("中年人")
    }
    // 在原本的 if 条件中增加了 else 的隐含条件
    if (age > 20 && !(age > 60) && !(age > 40 && !(age >60))) {
        println("青年人")
    }
}
```

上面程序的判断逻辑即转为如下三种情形。

- age 大于 60 岁，判断为“老年人”。
- age 大于 40 岁，且 age 小于或等于 60 岁，判断为“中年人”。
- age 大于 20 岁，且 age 小于或等于 40 岁，判断为“青年人”。

上面的判断逻辑才是实际希望看到的。因此，当使用 if else 语句进行流程控制时，一定不要忽略了 else 所带的隐含条件。

如果每次都去计算 if 条件和 else 条件的交集也是一件非常烦琐的事情，为了避免出现上面的错误，在使用 if else 语句时有一条基本规则：总是优先把包含范围小的条件放在前面处理。如 age>60 和 age>20 两个条件，明显 age>60 的范围更小，所以应该先处理 age>60 的情况。



注意：

在使用 if else 语句时，一定要先处理包含范围更小的情况。

4.2.2 if 表达式

Kotlin 的 if 分支还可作为表达式使用。也就是说，整个 if 表达式（包括 else 部分）最终会返回一个值，因此 if 可以代替 Java 中的三目运算符。

例如如下代码。

程序清单：codes\04\4.2\IfExpr.kt

```
fun main(args: Array<String>) {
    var age = 20
    // 将 if 表达式赋值给 str 变量
    var str = if (age > 20) "age 大于 20" else if(age < 20)
        "age 小于 20" else "age 等于 20"
    println(str)
}
```

上面粗体字代码示范了将 if 表达式的值赋给 str 变量的情形。从上面代码可以看出，当使用 if 分支作为表达式时，由于 if 表达式必须要有返回值，因此 if 表达式需要有 else 部分。

if 表达式的分支依然可以是用花括号括起来的代码块，此时代码块的最后一个表达式的值将作为整个代码块的值。例如如下代码。

程序清单：codes\04\4.2\IfExpr2.kt

```
fun main(args: Array<String>) {
    var age = 18
    // 将 if 表达式赋值给 str 变量
    var str = if (age > 20) {
        println("大于 20 的分支")
        "age 大于 20"
    } else if(age < 20) {
        println("小于 20 的分支")
        "age 小于 20"
    } else {
        println("else 的分支")
        "age 等于 20"
    }
    println(str)
}
```

对于上面程序而言，由于 age 的值为 18，因此 if 表达式将会执行 age < 20 分支对应的代码块，该代码块最后一个表达式为字符串："age 大于 20"，因此这个字符串将作为该代码块的值被赋给 str 变量。该程序的执行结果如下：

```
小于 20 的分支
age 小于 20
```

4.2.3 when 分支语句

when 分支取代了 Java 原有的 switch 语句，例如如下 Java 程序。

程序清单：codes\04\4.2\SwitchTest.java

```
public class SwitchTest{
    public static void main(String[] args) {
```

```
        char score = 'B';
        switch(score) {
            case 'A':
                System.out.println("优秀");
                break;
            case 'B':
                System.out.println("良好");
                break;
            case 'C':
                System.out.println("中");
                break;
            case 'D':
                System.out.println("及格");
                break;
            default :
                System.out.println("不及格");
        }
    }
}
```

将上面程序改为使用 when 分支，将会变得更加简洁。程序如下。

程序清单：codes\04\4.2\WhenTest.kt

```
fun main(args: Array<String>){
    var score = 'B'
    when (score){
        'A' -> println("优秀")
        'B' -> println("良好")
        'C' -> println("中")
        'D' -> println("及格")
        else -> println("不及格")
    }
}
```

对于上面两个程序，不难发现 when 分支其实就是 switch 分支的简化。when 分支的改变有以下几点。

- 不再需要使用 case 关键字。
- case 值后的冒号改为使用箭头（->）。
- default 改为更有意义、更明确的 else。



提示：

正如《疯狂 Java 讲义》中所提到的，switch 分支的 default 是很混淆的——default 字面意思是“默认”，但 switch 分支中的 default 其实代表“否则”的意思，when 分支将它改为 else 终于解除了笔者多年的尴尬。

如果 when 分支包含多条语句，则需要使用花括号将这些语句括成一个整体形成代码块。例如如下程序。

程序清单：codes\04\4.2\WhenTest2.kt

```
fun main(args: Array<String>){
    var score = 'B'
    when (score){
        'A' -> {
            println("优秀")
            println("望百尺竿头更进一步")
        }
        'B' -> {
```

```
            println("良好")
            println("不拼一把，你不知道自己的能力")
        }
        'C' -> println("中")
        'D' -> println("及格")
        else -> {
            println("不及格")
            println("啥也不说了，下次再来")
        }
    }
}
```

上面 when 语句的'A'、'B'、else 分支都包含两条语句，因此要使用花括号将这些语句括起来形成代码块。运行上面程序，可看到如下输出：

```
良好
不拼一把，你不知道自己的能力
```

如果 when 分支只是 switch 分支的简化，那也不过如此。事实上，when 分支比 switch 分支更强大，下面是 when 分支的 3 个小改进。

- when 分支可以匹配多个值。
- when 分支后的值不要求是常量，可以是任意表达式。
- when 分支对条件表达式的类型没有任何要求。

下面先看 when 分支匹配多个值的程序。

程序清单：codes\04\4.2\WhenMultiValue.kt

```
fun main(args: Array<String>){
    var score = 'B'
    when (score){
        'A', 'B' -> {
            println("优秀")
            println("望百尺竿头更进一步")
        }
        'C', 'D' -> println("中")
        else -> {
            println("不及格")
            println("啥也不说了，下次再来")
        }
    }
}
```

从上面代码可以看出，当 when 分支的条件表达式为'A'、'B'时，都会执行第一个分支；当 when 分支的条件表达式为'C'、'D'时，都会执行第二个分支。运行上面程序，可看到如下输出：

```
优秀
望百尺竿头更进一步
```

when 分支后的值不再要求是常量或字面值，可以是任意表达式。例如如下程序。

程序清单：codes\04\4.2\WhenTest3.kt

```
fun main(args: Array<String>){
    var score = 'B'
    var str = "EFGH"
    when (score){
        str[0] - 4, str[1] - 4  -> {
            println("优秀")
            println("望百尺竿头更进一步")
        }
        str[2] - 4, str[3] - 4 -> println("中")
```

```
        else -> {
            println("不及格")
            println("啥也不说了，下次再来")
        }
    }
}
```

从上面程序中的粗体字代码可以看到，各分支的值既不是常量，也不是字面值，而是一个表达式，这在 Kotlin 的 when 分支中完全合适。运行上面程序，可看到如下结果：

```
优秀
望百尺竿头更进一步
```

此外，when 分支不再对条件表达式的类型有任何要求，when 分支的条件表达式可以是任意类型。例如如下程序。

程序清单：codes\04\4.2\WhenTypeTest.kt

```
import java.util.Date
fun main(args: Array<String>){
    var date = Date()
    // when 分支的条件表达式是 Date 类型
    when (date){
        Date() -> {
            println("优秀")
            println("望百尺竿头更进一步")
        }
        else -> {
            println("不及格")
            println("啥也不说了，下次再来")
        }
    }
}
```

从上面的粗体字代码可以看出，此时 when 分支的条件表达式是 Date 对象，但这不影响 when 分支的执行，只要 when 的条件表达式与某个分支的值通过“==”比较返回 true，程序就可以进入执行该分支的代码。

4.2.4 when 表达式

与 if 分支相同，when 分支也可作为表达式。

如果 when 分支被当作表达式，那么符合条件的分支的代码块的值就是整个表达式的值。与 if 分支相同的是，如果分支的执行体是一个代码块，那么该代码块的值就是块中最后的表达式的值。

当 when 语句作为表达式使用时，when 表达式也需要有一个返回值，因此 when 表达式通常必须有 else 分支，除非编译器能够检测出所有的可能情况都已经被覆盖了。

如下代码示范了 when 表达式的用法。

程序清单：codes\04\4.2\WhenExpr.kt

```
fun main(args: Array<String>){
    var score = 'B'
    val str = when (score) {
        'A' -> {
            println("望百尺竿头更进一步")
            "优秀"
        }
        'B' -> {
```

```
            println("不拼一把，你不知道自己的能力")
            "良好"
        }
        'C' -> "中"
        'D' -> "及格"
        else -> {
            println("啥也不说了，下次再来")
            "不及格"
        }
    }
    println(str)
}
```

上面程序示范了 when 表达式的用法。在 when 表达式的每个分支的代码块的最后都定义了一个表达式作为该代码块的值，这样即可保证 when 表达式可以返回有效的值。运行上面程序，可看到如下输出：

```
不拼一把，你不知道自己的能力
良好
```

上面输出的第一行是执行 when 表达式中特定分支的结果；第二行输出是 when 表达式的返回值。

4.2.5 when 分支处理范围

通过使用 in、!in 运算符，我们还可以使用 when 分支检查表达式是否位于指定区间或集合中。例如如下代码。

程序清单：codes\04\4.2\WhenRange.kt

```
fun main(args: Array<String>){
    val age = java.util.Random().nextInt(100)
    println(age)
    // 使用 when 表达式对 str 赋值
    var str = when (age){
        in 10..25 -> "当时年少青衫薄"
        in 26..50 -> "风景依稀似去年"
        in 51..80 -> "醉听清吟胜管弦"
        else -> "其他"
    }
    println(str)
}
```

上面程序使用 when 表达式对 age 变量进行判断，该 when 表达式不再要求 age 等于某个具体的值，而是只要 age 处于特定范围中，即可进入相应的分支。运行该程序，可看到如下输出：

```
39
风景依稀似去年
```

从上面的运行结果可以看出，当 age 变量为 39 时，它处于 26..50 之间，when 表达式将会返回该分支的值。

4.2.6 when 分支处理类型

通过使用 is、!is 运算符，我们还可以使用 when 分支检查表达式是否为指定类型。例如如下代码。

程序清单：codes\04\4.2\WhenTypeDetect.kt

```
fun main(args: Array<String>){
    var inputPrice = 26
    println(realPrice(inputPrice))
}
// 程序对 inputPrice 的类型进行判断
fun realPrice(inputPrice: Any) = when(inputPrice){
    // 如果 inputPrice 的类型为 String，程序返回该字符串转换的 Double 值
    is String -> inputPrice.toDouble()
    // 如果 inputPrice 的类型为 Int，程序返回该 Int 值转换的 Double 值
    is Int -> inputPrice.toDouble()
    is Double -> inputPrice
    else -> 0.0
}
```

上面程序使用 when 表达式对 inputPrice 的类型进行判断。程序中 inputPrice 参数可能是任意类型，因此程序判断如果 inputPrice 是 String 类型，则调用 String 对象的 toDouble()将其转换为 Double 后返回；如果 inputPrice 是 Int 类型，则调用 Int 的 toDouble()将其转换为 Double 后返回；如果 inputPrice 是 Double 类型，则直接返回 inputPrice。

4.2.7 when 条件分支

when 分支还可以用来取代 if...else if 链，此时不需要为 when 分支提供任何条件表达式，每个分支条件都是一个布尔表达式，当指定分支的布尔表达式为 true 时执行该分支。例如如下代码。

程序清单：codes\04\4.2\WhenFork.kt

```
fun main(args: Array<String>){
    // 读取一行输入
    val ln = readLine()
    // ln 是 String?类型，所以需要先判断 ln 不为 null
    if(ln != null){
        // when 分支不需要任何条件表达式
        when {
            // 每个分支条件都是布尔表达式
            ln.matches(Regex("\\d+")) -> println("您输入的全是数字")
            ln.matches(Regex("[a-zA-Z]+")) -> println("您输入的全是字母")
            ln.matches(Regex("[a-zA-Z0-9]+")) -> println("您输入的是字母和数字")
            else -> println("您输入的包含特殊字符")
        }
    }
}
```

从上面代码可以看出，此时的 when 分支不需要任何条件表达式，每个分支都是一个布尔表达式，当某个分支的布尔表达式为 true 时，when 将会执行该分支的代码。

运行上面程序，程序将会根据用户输入反馈不同的输出。程序输出如下：

```
124327
您输入的全是数字
```

4.3 循环结构

循环语句可以在满足循环条件的情况下，反复执行某一段代码，这段被重复执行的代码被称为循环体。当反复执行这个循环体时，需要在合适的时候把循环条件改为假，从而结束循环；否则循环将一直执行下去，形成死循环。循环语句可能包含如下 4 个部分。

- **初始化语句**（init_statements）：一条或多条语句，用于完成一些初始化工作。初始化语句在循环开始之前执行。
- **循环条件**（test_expression）：这是一个布尔表达式，这个表达式能决定是否执行循环体。
- **循环体**（body_statements）：这个部分是循环的主体，如果循环条件允许，这个代码块将被重复执行。如果这个代码块只有一行语句，则这个代码块的花括号是可以省略的。
- **迭代语句**（iteration_statements）：这个部分在一次循环体执行结束后，对循环条件求值之前执行，通常用于控制循环条件中的变量，使得循环在合适的时候结束。

上面4个部分只是一般分类，并不是每个循环中都非常清晰地分出这4个部分。

4.3.1 while循环

while循环的语法格式如下：

```
[init_statements]
while test_expression {
    body_statements
    [iteration_statements]
}
```

while循环在每次执行循环体之前，都要先对test_expression循环条件求值，如果循环条件为真，则运行循环体部分。从上面的语法格式来看，迭代语句iteration_statements总是位于循环体的最后，因此只有当循环体能成功执行完成时，while循环才会执行迭代语句iteration_statements。

从这个意义上看，while循环也可被当成条件语句——如果test_expression条件一开始就为假，则循环体部分将永远不会获得执行的机会。

下面程序示范了一个简单的while循环。

程序清单：codes\04\4.3\WhileTest.kt

```
fun main(args: Array<String>) {
    // 循环的初始化条件
    var count = 0
    // 当count小于10时，执行循环体
    while (count < 10) {
        println("count:${count}")
        // 迭代语句
        count++
    }
    println("循环结束!")
}
```

使用while循环时，一定要保证循环条件有变成假的时候；否则这个循环将成为一个死循环，永远无法结束这个循环。例如如下代码（程序清单同上）：

```
// 下面是一个死循环
var count2 = 0
while (count2 < 10) {
    print("不停执行的死循环 ${count2}")
    count2--
}
println("永远无法跳出的循环体")
```

在上面代码中，count2的值越来越小，这将导致count2的值永远小于10，count2 < 10循

环条件一直为 true，从而导致这个循环永远无法结束。

与 Java 类似的是，如果 while 循环的循环体只有一行语句，那么 Kotlin 允许省略循环体的花括号。例如如下程序。

程序清单：codes\04\4.3\WhileNoEmbrace.kt

```
fun main(args: Array<String>) {
    // 循环的初始化条件
    var count = 0
    // 当 count 小于 10 时，执行循环体
    while (count < 10)
        println("count:${count++}")
}
```

与前面 if 分支类似的是，如果省略了循环体的花括号，那么 while 循环只控制紧跟该循环的一条语句。在极端情况下，如果 while 循环后面有一个分号（分号相当于一条空语句），此时 while 循环将仅控制这条空语句作为循环体。

4.3.2 do while 循环

do while 循环与 while 循环的区别在于：while 循环是先判断循环条件，如果条件为真则执行循环体；而 do while 循环则先执行循环体，然后才判断循环条件，如果循环条件为真，则执行下一次循环，否则中止循环。do while 循环的语法格式如下：

```
[init_statements]
do
{
    body_statements
    [iteration_statements]
}while (test_expression)
```

下面程序示范了 do while 循环的用法。

程序清单：codes\04\4.3\DoWhileTest.kt

```
fun main(args: Array<String>) {
    // 定义变量 count
    var count = 1
    // 执行 do while 循环
    do {
        println(count)
        // 循环迭代语句
        count++
        // 循环条件紧跟 while 关键字
    } while (count < 10)
    println("循环结束!")
}
```

即使 test_expression 循环条件的值开始就是假，do while 循环也会执行循环体。因此，do while 循环的循环体至少执行一次。下面的代码片段验证了这个结论（程序清单同上）。

```
// 定义变量 count2
var count2 = 20
// 执行 do while 循环
do
    // 这行代码把循环体和迭代部分合并成一行代码
    println(count2++)
while (count2 < 10)
println("循环结束!")
```

从上面程序来看，虽然开始 count2 的值就是 20，count2 < 10 表达式返回 false，但 do while 循环还是会执行一次循环体。

4.3.3 for-in 循环

for-in 循环专门用于遍历范围、序列和集合等包含的元素。for-in 循环的语法格式如下：

```
for (常量名 in 字符串|范围|集合) {
    statements
}
```

对于上面语法格式有两点说明。

- for-in 循环中的常量无须声明。for-in 循环中的常量将会在每次循环开始时自动被赋值，因此该常量无须提前声明。只要将它包含在 for-in 循环的声明中，即可隐式声明该常量，无须使用 let 关键字声明。
- for-in 循环可用于遍历任何可迭代对象。所谓可迭代对象就是该对象包含一个 iterator() 方法，且该方法的返回值对象具有 next()、hasNext()方法，这三个方法都使用 operator 修饰。

for-in 循环可用于遍历范围。例如，如下程序使用 for-in 循环来计算指定整数的阶乘。

程序清单：codes\04\4.3\ForinRange.swift

```
fun main(args: Array<String>) {
    var max = 7
    var result = 1
    // 使用 for-in 循环遍历范围
    for( num in 1..max ) {
        result *= num
    }
    println(result)
}
```

for-in 循环的循环计数器相当于一个用 val 声明的常量，因此程序不允许在 for-in 循环中对循环计数器进行赋值。例如如下程序（程序清单同上）：

```
for( i in 1 until 5) {
    println("i: ${i}")
    // 对 for-in 循环的循环计数器赋值会导致错误
    i = 20
}
```

上面代码中的粗体字赋值语句会导致如下编译提示：

```
ForinRange.kt:24:3: error: val cannot be reassigned
```

通过上面错误提示可以看出，for-in 循环的循环计数器相当于用 val 声明的常量，因此不允许被重新赋值。

除此之外，for-in 循环还可用于遍历数组、List、字典等集合。关于 for-in 循环遍历数组、List、字典的元素相关知识，本书将在下一章中介绍。

4.3.4 嵌套循环

如果把一个循环放在另一个循环体内，那么就可以形成嵌套循环。嵌套循环既可以是 for-in 循环嵌套 while 循环，也可以是 while 循环嵌套 do while 循环……即各种类型的循环都可以作为外层循环，各种类型的循环也都可以作为内层循环。

当程序遇到嵌套循环时，如果外层循环的循环条件允许，则开始执行外层循环的循环体，

而内层循环将被外层循环的循环体来执行——只是内层循环需要反复执行自己的循环体而已。当内层循环执行结束，且外层循环的循环体执行结束后，将再次计算外层循环的循环条件，决定是否再次开始执行外层循环的循环体。

根据上面分析，假设外层循环的循环次数为 n 次，内层循环的循环次数为 m 次，那么内层循环的循环体实际上需要执行 $n \times m$ 次。嵌套循环的执行流程如图 4.1 所示。

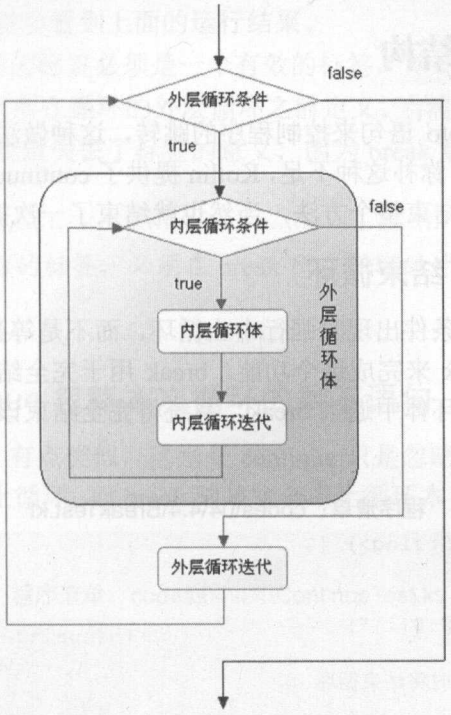


图 4.1 嵌套循环的执行流程

从图 4.1 来看，嵌套循环就是把内层循环当成外层循环的循环体。只有内层循环的循环条件为假时，才会完全跳出内层循环，才可以结束外层循环的当次循环，开始下一次循环。下面是一个嵌套循环的示例代码。

程序清单：codes\04\4.3\NestedLoopTest.kt

```
fun main(args: Array<String>) {
    // 外层循环
    for( i in 0 until 5 ){
        var j = 0
        // 内层循环
        while( j < 3 ) {
            println("i 的值为: ${i} , j 的值为: ${j++}")
        }
    }
}
```

运行上面程序，将看到如下运行结果：

```
i 的值为:0  j 的值为:0
i 的值为:0  j 的值为:1
i 的值为:0  j 的值为:2
......
```

从上面的运行结果可以看出，进入嵌套循环时，循环变量 i 开始为 0，这时即进入了外层

循环。进入外层循环后，内层循环把 i 当成一个普通变量，其值为 0。在外层循环的当次循环中，内层循环就是一个普通循环。

实际上，嵌套循环不仅可以是两层嵌套，还可以是三层嵌套、四层嵌套……不论循环如何嵌套，都可以把内层循环当成外层循环的循环体来对待，区别只是这个循环体里包含了需要反复执行的代码。

4.4 控制循环结构

Kotlin 语言没有提供 goto 语句来控制程序的跳转，这种做法提高了程序流程控制的可读性，但降低了其灵活性。为了弥补这种不足，Kotlin 提供了 continue 和 break 来控制循环结构。除此之外，使用 return 可以结束整个方法，当然也就结束了一次循环。

4.4.1 使用 break 结束循环

某些时候，需要在某种条件出现时强行中止循环，而不是等到循环条件为 false 时才退出循环。此时，可以使用 break 来完成这个功能。break 用于完全结束一个循环，跳出循环体。不管是哪种循环，一旦在循环体中遇到 break，系统将完全结束该循环，开始执行循环之后的代码。例如如下程序。

程序清单：codes\04\4.4\BreakTest.kt

```
fun main(args: Array<String>) {
    // 一个简单的 for 循环
    for (i in 0 .. 10 ) {
        println("i 的值是:${i}")
        if (i == 2) {
            // 执行该语句时将结束循环
            break
        }
    }
}
```

运行上面程序，将看到 i 循环到 2 时即结束，当 i 等于 2 时，在循环体内遇到 break 语句，程序跳出该循环。

使用 break 语句不仅可以结束其所在的循环，还可以直接结束其外层循环。此时需要在 break 后紧跟一个标签，这个标签用于标识一个外层循环。

Kotlin 中的标签就是一个紧跟着@的标识符。Kotlin 中的标签只有放在循环语句或 switch 语句之前才起作用。例如下面代码。

程序清单：codes\04\4.4\BreakLabel.kt

```
fun main(args: Array<String>) {
    // 外层循环，outer 作为标识符
    outer@ for (i in 0 until 5 ) {
        // 内层循环
        for (j in 0 until 3 ) {
            println("i 的值为:${i}, j 的值为:${j}")
            if (j == 1) {
                // 跳出 outer 标签所标识的循环
                break@outer
            }
        }
    }
}
```

运行上面程序，将看到如下运行结果：

```
i 的值为:0, j 的值为:0
i 的值为:0, j 的值为:1
```

程序从外层循环进入内层循环后，当 j 等于 1 时，程序遇到一条 break@outer 语句，这条语句将会导致结束 outer@标签指定的循环——不是结束 break 所在的循环，而是结束 outer@标签所标识的外层循环。所以会看到上面的运行结果。

值得指出的是，break 后的标签必须是一个有效的标签，即这个标签必须在 break 语句所在的循环之前定义，或者在其所在循环的外层循环之前定义。当然，如果把这个标签放在 break 语句所在的循环之前定义，也就失去了标签的意义，因为 break 默认就是结束其所在的循环。

注意：

通常紧跟 break 之后的标签，必须在 break 所在循环的外层循环之前定义才有意义。



4.4.2 使用 continue 忽略本次循环的剩下语句

continue 的功能和 break 有点类似，区别是 continue 只是忽略本次循环的剩下语句，接着开始下一次循环，并不会中止循环；而 break 则是完全中止循环本身。如下程序示范了 continue 的用法。

程序清单：codes\04\4.4\ContinueTest.kt

```
fun main(args: Array<String>) {
    // 一个简单的 for 循环
    for (i in 0 until 3 ) {
        println("i 的值是${i}")
        if (i == 1) {
            // 忽略本次循环的剩下语句
            continue
        }
        println("continue 后的输出语句")
    }
}
```

运行上面程序，将看到如下运行结果：

```
i 的值是 0
continue 后的输出语句
i 的值是 1
i 的值是 2
continue 后的输出语句
```

从上面的运行结果来看，当 i 等于 1 时，程序没有输出“continue 后的输出语句”字符串，因为程序执行到 continue 时，忽略了当次循环中 continue 语句后的代码。从这个意义上看，如果把一条 continue 语句放在当次循环的最后一行，那么这条 continue 语句是没有任何意义的——因为它仅仅忽略了一片空白，没有忽略任何程序语句。

与 break 类似的是，continue 后也可以紧跟一个标签，用于直接跳过标签所标识循环的当次循环的剩下语句，重新开始下一次循环。例如下面代码。

程序清单：codes\04\4.4\ContinueLabel.kt

```
fun main(args: Array<String>) {
    // 外层循环
    outer@ for( i in 0 until 5 ){
        // 内层循环
        for( j in 0 until 3 ){
            println("i 的值为:${i}, j 的值为:${j}")
            if (j == 1) {
                // 忽略 outer 标签所指定的循环中当次循环剩下的语句
                continue@outer
            }
        }
    }
}
```

运行上面程序可以看到，循环变量j的值将无法超过1，因为每当j等于1时，continue@outer语句就结束了外层循环的当次循环，直接开始下一次循环，内层循环没有机会执行完成。

与break类似的是，continue后的标签也必须是一个有效的标签，即这个标签通常应该放在continue所在循环的外层循环之前定义。

4.4.3 使用return结束方法

return用于从最直接包围它的方法、函数或者匿名函数返回。当函数或方法执行到一条return语句时（return关键字后还可以跟变量、常量和表达式，在函数或方法介绍中将有更详细的解释），这个函数或方法将被结束。

Kotlin程序中的大部分循环都被放在函数或方法中执行，一旦在循环体内执行到一条return语句时，return语句就会结束该函数或方法，循环自然也随之结束。例如下面程序。

程序清单：codes\04\4.4\ReturnTest.kt

```
fun main(args: Array<String>) {
    test()
}
fun test() {
    // 一个简单的 for 循环
    for( i in 0 until 10) {
        println("i 的值是: ${i}")
        if ( i == 1) {
            return
        }
        println("return 后的输出语句")
    }
}
```

运行上面程序，循环只能执行到i等于1时，当i等于1时程序将完全结束（当test()函数结束时，也就是Kotlin程序结束时）。从这个运行结果来看，虽然return并不是专门用于控制循环结构的关键字，但通过return语句确实可以结束一个循环。与continue和break不同的是，return直接结束整个函数或方法，而不管return处于多少层循环之内。



提示：

上面代码通过函数语法来定义函数，读者可参考本书第6章来了解函数有关知识。

4.5 本章小结

本章主要介绍了Kotlin的两种程序流程结构：分支结构和循环结构。本章详细讲解了Kotlin提供的if和when分支结构，Kotlin的分支不仅可作为语句使用，也可作为表达式用于赋值，因此使用起来非常灵活。Kotlin语言的when语句功能非常强大，不仅可以匹配简单值，还可以匹配范围，判断类型，甚至可以完全替代if分支，希望读者好好掌握when分支的用法。本章还详细介绍了Kotlin提供的while、do while和for-in循环结构，并详细分析了这三种循环结构的区别和联系。

CHAPTER

5

第5章 数组和集合

本章要点

- 声明和创建数组
- 使用和操作数组元素
- 使用 for-in 遍历数组
- 使用数组索引
- 多维数组
- 数组的常用应用
- Kotlin 集合的设计和功能
- 声明和创建 Set
- 使用和操作 Set 集合元素
- 遍历 Set
- 使用可变 Set
- 声明和创建 List
- 使用和操作 List 集合元素
- 使用可变 List
- 声明和创建 Map
- 使用和操作 Map 集合元素
- 遍历 Map
- 使用可变 Map

Kotlin 为数组增加了一个 Array 类，为元素是基本类型的数组增加了 XxxArray 类（其中 Xxx 可以是 Byte、Short、Int 等基本类型），因此开发者完全可用面向对象的语法来使用 Kotlin 的数组，包括创建数组对象、调用数组对象的属性和方法等。

Kotlin 自己提供了一套集合体系，Kotlin 的集合体系抛弃了 Java 集合体系中的 Queue 集合，但增加了可变集合和不可变集合的概念。Kotlin 的集合体系由三种集合组成，如图 5.1 所示。

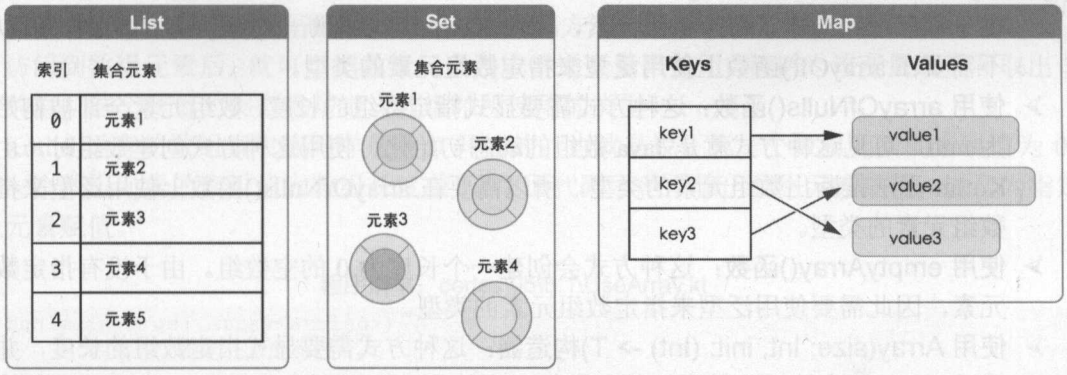


图 5.1 Kotlin 的集合体系

与 Java 集合类似的是，Kotlin 的 List 代表有序、集合元素可重复的集合；Set 代表无序、集合元素不可重复的集合；Map 则采用 key-value 对的形式存储数据，每项数据都由 key-value 对组成。

5.1 数组

Kotlin 的数组使用 Array<T>类代表，由此可见，Kotlin 数组就是一个 Array 类的实例，因此 Kotlin 数组当然也算引用类型。

5.1.1 创建数组

Kotlin 创建数组其实就是创建 Array<T>类的实例，Kotlin 既可允许通过 Array<T>类的构造器来创建实例，也可通过 arrayOf()等工具函数来创建实例。

Kotlin 创建数组大致有如下两种方式。

- 使用 arrayOf()、arrayOfNulls()、emptyArray()工具函数。
- 使用 Array(size: Int, init: (Int) -> T)构造器。

下面代码示范了 Kotlin 创建数组。

程序清单：codes\05\5.1\InitArray.kt

```
fun main(args: Array<String>) {
    // 创建包含指定元素的数组（相当于 Java 数组的静态初始化）
    var arr1 = arrayOf("Java", "Kotlin", "Swift", "Go")
    var intArr1 = arrayOf(2, 4, 500, -34)
    // 创建指定长度、元素为 null 的数组（相当于 Java 数组的动态初始化）
    var arr2 = arrayOfNulls<Double>(5)
    var intArr2 = arrayOfNulls<Int>(6)
    // 创建长度为 0 的空数组
    var arr3 = emptyArray<String>()
    var intArr3 = emptyArray<Int>()
    // 创建指定长度、使用 Lambda 表达式初始化数组元素的数组
    var arr4 = Array(5, {(it * 2 + 97).toChar()})
```

```
    var strArr4 = Array(6, {"fkit"})
}
```

上面程序中的 8 行粗体字代码分别为每种创建方式提供了两行示范代码。下面简单分析 4 种创建方式。

- **使用 arrayOf()函数**：这种方式无须显式指定数组的长度，但需要依次列出每个数组元素。因此，这种方式其实就相当于 Java 数组的静态初始化。使用这种方式创建数组时，由于程序已经给出了每个数组元素，因此 Kotlin 可以推断出数组元素的类型。所以，不需要在 arrayOf()函数上使用泛型来指定数组元素的类型。
- **使用 arrayOfNulls()函数**：这种方式需要显式指定数组的长度，数组元素全部被初始化为 null，可见这种方式就是 Java 数组的动态初始化。使用这种方式创建数组时，由于 Kotlin 无法推断出数组元素的类型，所以需要在 arrayOfNulls()函数上使用泛型来指定数组元素的类型。
- **使用 emptyArray()函数**：这种方式会创建一个长度为 0 的空数组。由于没有指定数组元素，因此需要使用泛型来指定数组元素的类型。
- **使用 Array(size: Int, init: (Int) -> T)构造器**：这种方式需要显式指定数组的长度，并可通过 Lambda 表达式来动态计算各数组元素的值。这种方式是原来 Java 所不具备的。



提示：

上面程序的 Lambda 表达式中的 it 用于代表该表达式的唯一形参。关于 Lambda 表达式的介绍，可参考本书下一章的内容。

此外，由于 Array<T>类要求它的元素必须是引用类型，因此，如果程序将基本类型的值存入 Array<T>中，Kotlin 会将这些基本类型的值自动装箱成包装类的实例，然后才将这些实例添加到 Array<T>数组中。为此，Kotlin 专门提供了 ByteArray、ShortArray、IntArray、LongArray、CharArray、FloatArray、DoubleArray、BooleanArray，分别用于映射 Java 的 byte[]、short[]、int[]、long[]、char[]、float[]、double[]、boolean[]这 8 种基本类型的数组。

创建 XxxArray 对象的方式与前面介绍的方式大致相同。例如如下代码。

程序清单：codes\05\5.1\InitBasicArray.kt

```
import java.util.Arrays
fun main(args: Array<String>) {
    // 创建包含指定元素的数组（相当于 Java 数组的静态初始化）
    var intArr = intArrayOf(2, 4, 30, -5)
    var doubleArr = doubleArrayOf(2.3, 3.5, -3.0)
    // 创建指定长度、使用 Lambda 表达式初始化数组元素的数组
    var intArr2 = IntArray(5, {it * it})
    var charArr = CharArray(5, {(it * 2 + 97).toChar()})
    println(Arrays.toString(intArr2))
    println(Arrays.toString(charArr))
}
```

从上面代码可以看出，对于 XxxArray 数组，同样可采用列出数组元素（静态初始化）的方式来创建数组，但不支持使用将元素初始化为 null 的方式来创建数组——这是因为基本类型的元素不允许为 null。

运行上面程序，可看到如下输出：

```
[0, 1, 4, 9, 16]
[a, c, e, g, i]
```

5.1.2 使用数组

数组的常用方法就是访问数组元素，包括对数组元素进行赋值和取出数组元素的值。访问数组元素都是通过在数组引用变量后紧跟一个方括号（[]）实现的，方括号里是数组元素的索引值。正如前面所介绍的，Kotlin 的方括号运算符其实是 get(index)和 set(index, value)方法，因此，当程序使用[i]获取数组元素的值时，实际上就是调用 get(index)方法；当使用[index]对数组元素赋值时，实际上就是调用 set(index ,value)方法。

访问到数组元素后，就可以把数组元素当成普通变量来使用，包括为该变量赋值和取出该变量的值，这个变量的类型就是定义数组时所使用的类型。

Kotlin 语言的数组索引值同样是从 0 开始的。也就是说，第一个数组元素的索引值为 0，最后一个数组元素的索引值为数组长度减 1。下面代码示范了输出数组元素的值，以及为指定数组元素赋值。

程序清单：codes\05\5.1\UseArray.kt

```
fun main(args: Array<String>) {
    // 创建包含指定元素的数组
    var strArr = arrayOf("Kotlin", "Java", "Go", "Swift")
    // 使用两种方式获取数组元素
    println(strArr[1])
    println(strArr.get(1))
    // 使用两种方式修改数组元素
    strArr[1] = "Groovy"
    strArr.set(2, "Lua")
    println(java.util.Arrays.toString(strArr))
}
```

上面 4 行粗体字代码分别示范了访问数组元素和修改数组元素。访问、修改数组元素时既可使用[index]的方式来实现，也可使用 get(index)、set(index, value)的方式来实现。这两种方式的本质是一样的，而且编译器经过编译优化之后，还是会转换成根据数组的内存地址来访问数组元素，因此性能不会有任何损失。

运行上面程序，可看到如下输出：

```
Java
Java
[Kotlin, Groovy, Lua, Swift]
```

如果访问数组元素时指定的索引值小于 0，或者大于或等于数组的长度，那么编译程序时不会出现任何错误，但运行时会出现异常：java.lang.ArrayIndexOutOfBoundsException: N（数组索引越界异常），异常信息后的 N 就是程序员试图访问的数组元素索引值。

下面代码试图访问的数组元素索引值等于数组长度，将引发数组索引越界异常（程序清单同上）。

```
// 访问数组元素指定的索引值等于数组长度，所以下面代码将在运行时出现异常
println(strArr[4])
```

所有的数组都提供了一个 size 属性，通过这个属性可以访问到数组的长度，一旦获得了数组的长度，就可以通过循环来遍历该数组的每个数组元素。下面代码示范了输出 prices 数组的每个数组元素的值（程序清单同上）。

```
// 创建指定长度、元素为 null 的数组
var prices = arrayOfNulls<Double>(5)
// 使用循环输出 prices 数组的每个数组元素的值
```

```
for (i in 0 until prices.size ) {
    println(prices[i])
}
```

执行上面代码，将输出 5 个 null。

下面代码示范了为动态初始化的数组元素进行赋值，并通过循环方式输出每个数组元素的值（程序清单同上）。

```
var books = arrayOfNulls<String>(4)
// 对动态初始化后的数组元素进行赋值
books[0] = "疯狂 Java 讲义"
books[1] = "轻量级 Java EE 企业应用实战"
// 使用循环输出 books 数组的每个数组元素的值
for (i in 0 until books.size ){
    println(books[i])
}
```

上面代码将先输出字符串"疯狂 Java 讲义"和"轻量级 Java EE 企业应用实战"，然后输出两个 null，因为 books 被初始化为长度为 4、元素都是 null 的 Array<String>数组，接下来程序又为前两个元素赋值，所以看到了这样的程序输出结果。

从上面代码中不难看出，初始化一个数组后，相当于同时初始化了多个相同类型的变量，通过数组元素的索引值就可以自由访问这些变量（实际上都是数组元素）。使用数组元素与使用普通变量并没有什么不同，一样可以对数组元素进行赋值，或者取出数组元素的值。

5.1.3 使用 for-in 循环遍历数组

Kotlin 的 for-in 循环可自动遍历数组的每个元素。下面程序示范了如何使用 for-in 循环来遍历数组元素。

程序清单：codes\05\5.1\ForinTest.kt

```
fun main(args: Array<String>) {
    var books = arrayOf("轻量级 Java EE 企业应用实战" ,
        "疯狂 Java 讲义",
        "疯狂 Android 讲义")
    // 使用 for-in 循环来遍历数组元素
    // 其中 book 将会自动迭代每个数组元素
    for (book in books) {
        println(book)
    }
}
```

使用 for-in 循环来遍历数组元素或集合元素时，不允许对循环变量进行赋值。

对数组使用 for-in 循环依然会被编译成使用基于索引的循环，并不会创建迭代器进行遍历，因此使用 for-in 循环遍历数组依然具有很好的性能。

5.1.4 使用数组索引

前面使用索引遍历数组时还使用了 0 until array.size 这种方式来创建区间，其实完全没这个必要，因为 Kotlin 的数组还提供了一个 indices 属性，该属性即可返回数组的索引区间。例如，如下程序利用 indices 属性返回的区间来遍历数组。

程序清单：codes\05\5.1\IndicesTest.kt

```
fun main(args: Array<String>) {
    var books = arrayOf("轻量级 Java EE 企业应用实战" ,
        "疯狂 Java 讲义",
```

```
        "疯狂 Android 讲义")
    // 根据数组的索引来遍历数组
    for (i in books.indices) {
        println(book[i])
    }
}
```

这种通过索引区间遍历的实现具有更好的性能，因此 Kotlin 将会在底层将其编译成根据内存地址来访问数组元素，因此不需要额外创建区间对象。

此外，程序还可使用 in、!in 来检查某个索引是否位于数组的索引区间内。例如如下代码（程序清单同上）：

```
var i = java.util.Random().nextInt(10)
// 检查 i 是否位于 books 数组的索引区间内
println( i in books.indices) // i >=0 && i < books.size
```

另外，Kotlin 还为数组提供了一个 lastIndex 属性，该属性用于返回数组最后一个元素的索引值，该索引值通常等于 size - 1。例如如下代码（程序清单同上）：

```
println( books.lastIndex) // 输出 2
println( books.size - 1 == books.lastIndex) // 输出 true
```

如果程序需要同时访问数组的索引和元素，则也可使用数组的 withIndex()方法，该方法返回一个 Iterable 对象，该对象的所有元素都是 IndexedValue。例如，如下代码可同时访问数组的索引和元素。

程序清单：codes\05\5.1\WithIndexTest.kt

```
fun main(args: Array<String>) {
    var books = arrayOf("轻量级 Java EE 企业应用实战" ,
        "疯狂 Java 讲义",
        "疯狂 Android 讲义")
    // 通过 withIndex()方法可同时访问数组的索引和元素
    for((index, value) in books.withIndex()){
        println("索引为${index}的元素是：${value}")
    }
}
```

5.1.5 数组的常用方法

Kotlin 的数组是由 Array、XxxArray 这些类代表的，而 Kotlin 为这些类提供了大量的工具方法（例如前面介绍的 withIndex()方法），程序只要使用这些方法即可非常方便地操作数组。



提示：

Kotlin 为数组提供的方法大致相当于 Java 的 Arrays 工具类所提供的操作数组的方法。

下面仅仅列出 Array 类提供的常用工具方法。关于该类更详细的方法列表，可参考 https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-array/index.html 页面。

- all(predicate: (T) -> Boolean)：使用 Lambda 表达式要求所有数组元素都满足该表达式，如果都满足，那么该方法返回 true。
- any(predicate: (T) -> Boolean)：使用 Lambda 表达式要求任一数组元素都满足该表达式，如果都满足，那么该方法返回 true。
- asList()：将该数组转换成 List 集合。

- associate(transform: (T) -> Pair<K, V>)：使用 Lambda 表达式根据数组元素进行计算，返回元素是<K, V>的 Map 集合。
- associateBy(keySelector: (T) -> K)：使用 Lambda 表达式根据数组元素进行计算，返回元素是<K, T>的 Map 集合。
- associateBy(keySelector: (T) -> K, valueTransform: (T) -> V)：使用 Lambda 表达式根据数组元素进行计算，返回元素是<K, V>的 Map 集合。
- associateByTo(dest: M, keySelector: (T) -> K)：使用 Lambda 表达式根据数组元素进行计算，并将计算得到的<K, T>对添加到可变 Map 集合中，该方法返回被修改的 dest 集合。
- associateByTo(dest: M, keySelector: (T) -> K, valueTransform: (T) -> V)：使用 Lambda 表达式根据数组元素进行计算，并将计算得到的<K, V>对添加到可变 Map 集合中，该方法返回被修改的 dest 集合。
- associateTo(dest: M, transform: (T) -> Pair<K, V>)：使用 Lambda 表达式根据数组元素进行计算，并将计算得到的<K, V>对添加到可变 Map 集合中，该方法返回被修改的 dest 集合。
- average()：计算数值型数组的平均值。
- binarySearch(element: T, comparator: Comparator<in T>, fromIndex: Int = 0, toIndex: Int = size)：使用二分法查询 element 元素值在数组中出现的索引；如果数组不包含 element 元素值，则返回负数。调用该方法时要求数组中元素已经按升序排列，这样才能得到正确的结果。
- int binarySearch(element: T, fromIndex: Int = 0, toIndex: Int = size)：这个方法与前一个方法类似，只是该方法要求数组元素按自然排序（数组元素要实现 Comparable 接口）升序排列。
- contains(element: T)：判断该数组是否包含某个元素，该方法可用 in、!in 运算符执行。
- contentEquals(other: Array<out T>)：比较两个数组是否相等。
- contentToString()：把数组转换成字符串，相当于 Arrays 类的 toString()方法。
- copyOf(newSize: Int)：这个方法将会把该数组复制成一个新数组，其中 newSize 是新数组的长度。如果 newSize 小于原数组的长度，则新数组就是原数组的前面 newSize 个元素；如果 newSize 大于原数组的长度，则新数组的前面元素就是原数组的所有元素，后面补充 0（数值类型）、false（布尔类型）或者 null（引用类型）。该方法有个重载的版本，允许不指定 newSize 参数，则新数组与原数组长度相同。
- copyOfRange(fromIndex: Int, toIndex: Int)：这个方法与前一个方法相似，但该方法只复制该数组的 from 到 to 索引的元素。
- count(predicate: (T) -> Boolean)：返回该数组符合给定 Lambda 条件的元素的个数。该方法有个重载的版本，允许不指定 predicate 参数，此时直接返回该数组中元素的个数，相当于 size 属性的返回值。
- distinct()：该方法用于去掉数组中重复的元素。
- drop | dropLast(n: Int)：该方法用于去掉数组前面或后面 *n* 个元素。
- dropWhile | dropLastWhile(predicate: (T) -> Boolean)：该方法用于去掉数组前面或后面的某些元素，直到前面或后面第一个元素不再符合 predicate 条件。
- fill(element: T, fromIndex: Int = 0, toIndex: Int = size)：该方法将会把数组中从 fromIndex

到 toIndex 索引的数组元素赋值为 element。

- filterXxx()：一系列过滤方法，用于对数组元素进行过滤。
- findXxx()：一系列用于查找元素的方法。
- first | last(predicate: (T) -> Boolean)：获取数组中第一个或最后一个符合 predicate 条件的元素。
- fold(initial: R, operation: (acc: R, T)：将 initial、数组元素作为参数传入 operation 表达式执行计算，将计算得到的结果作为下一个数组元素的 initial，依此类推，直到使用最后一个数组元素计算，该方法返回使用最后一个数组元素计算得到的值。
- indexOf | lastIndexOf(element: T)：获取从前搜索或从后搜索时元素 element 在数组中的索引。
- indexOfFirst | indexOfLast(predicate: (T) -> Boolean)：返回第一个或最后一个符合 predicate 条件的元素的索引。
- intersect | plus(other: Iterable<T>)：获取两个数组的交集或并集。
- max|min()：按照自然排序规则（要求元素必须实现 Comparable 接口）找出该数组中最大值或最小值。
- sort(fromIndex: Int = 0, toIndex: Int = size)：该方法对数组的元素按自然排序（要求所有元素实现 Comparable 接口）进行排列。
- sortWith(comparator: Comparator<in T>)：该方法对数组的所有元素按 comparator 排序（定制排序）进行排列。

下面程序简单示范了 Array 类的工具方法的用法。

程序清单：codes\05\5.1\ArrayFun.kt

```
fun main(args: Array<String>) {
    // 定义一个数组
    var arr = arrayOf(2, 4, 5, 6)
    // 判断是否所有元素的平方都大于 20
    println(arr.all({it * it > 20})) // 输出 false
    // 判断是否任一元素的平方大于 20
    println(arr.any({it * it > 20})) // 输出 true
    // 根据数组元素来计算<K,V>对，返回所有<K,V>对组成的 Map 集合
    // 下面的算法规则是：K 是数组元素+2，V 是数组元素的平方
    var result1 = arr.associate({it + 10 to it * it})
    println(result1)
    // 创建一个可变 Map 集合，用于追加根据数组计算出来的 key-value 对
    var map = mutableMapOf(1 to 100, 2 to 120, -1 to 130)
    // 将计算出来的 key（元素的平方）、value（元素）对添加到 map 集合中
    arr.associateByTo(map, {it * it})
    println(map)
    // 计算数组所有元素的总和
    println(arr.fold(0 , {acc, e -> acc + e})) // 输出 17

    // 定义一个 a 数组
    var a = arrayOf(3, 4 , 5, 6)
    // 定义一个 a2 数组
    var a2 = arrayOf(3, 4 , 5, 6)
    // a 数组和 a2 数组的长度相等，每个元素依次相等，将输出 true
    println("a 数组和 a2 数组是否相等：${a.contentEquals(a2)}")
    // 通过复制 a 数组，生成一个新的 b 数组
    var b = a.copyOf(6)
    println("a 数组和 b 数组是否相等：${a.contentEquals(b)}")
    // 输出 b 数组的元素，将输出[3, 4, 5, 6, null, null]
```

```
    println("b 数组的元素为：${b.contentToString()}")
    // 将 b 数组的第 5 个元素（包括）到第 7 个元素（不包括）赋值为 1
    b.fill(1, 4, 6)
    // 输出 b 数组的元素，将输出[3, 4, 5, 6, 1, 1]
    println("b 数组的元素为：${b.contentToString()}")
    // 对 b 数组进行排序
    b.sort()
    // 输出 b 数组的元素，将输出[1, 1, 3, 4, 5, 6]
    println("b 数组的元素为：${b.contentToString()}")
}
```

由于数组还提供了用 operator 修饰的 plus()、contains()方法，因此数组可通过+、in 运算符进行运算。例如如下代码（程序清单同上）：

```
var c = arrayOf("Java", "Go", "Kotlin")
var d = arrayOf("Lua", "Kotlin", "Python")
println((c + d).contentToString()) // 计算两个集合的并集
println("Java" in c) // 输出 true
```

5.1.6 多维数组

表面上看，Kotlin 没有提供多维数组。如果读者对《疯狂 Java 讲义》中关于多维数组的内容有深刻理解，应该就会明白：所谓的多维数组其实都是一维数组，只要让数组的元素又是数组，那么该数组就变成了多维数组。

下面程序示范了如何用一维数组的方式来处理二维数组。

程序清单：codes\05\5.1\TwoDimensionTest.kt

```
fun main(args: Array<String>) {
    // 把 a 当成一维数组进行初始化，初始化 a 是一个长度为 4 的数组
    // a 数组的元素又是 Array<Int>类型
    var a = arrayOfNulls<Array<Int>>(4)
    // 把 a 数组当成一维数组，遍历 a 数组的每个数组元素
    for (i in a.indices){
        println(a[i])
    }
    // 初始化 a 数组的第一个元素
    a[0] = arrayOf(2, 5)
    // 访问 a 数组的第一个元素所指数组的第二个元素
    a[0]?.set(1 , 6)
    // a 数组的第一个元素是一个一维数组，遍历这个一维数组
    for (i in a[0]!!.indices ){
        println(a[0]?.get(i))
    }
}
```

上面程序中的粗体字代码把数组 a 当成一维数组处理，只是每个数组元素都是 null，所以可以看到输出结果都是 null。下面结合示意图来说明这个程序的执行过程。

程序中 var a = arrayOfNulls<Array<Int>>(4)这行代码让 a 变量指向一个长度为 4 的数组内存，这个长度为 4 的数组的每个元素都是数组类型，这些数组元素的初始值为 null。此时 a 数组在内存中的存储示意图如图 5.2 所示。

从图 5.2 来看，此时丝毫看不出数组 a 是一个二维数组的样子，完全是一维数组的样子。这个一维数组的长度是 4，只是这 4 个数组元素都是数组类型，且它们的初始值都是 null。所以在程序中可以把 a 数组当成一维数组处理，依次遍历 a 数组的每个数组元素，将看到每个数组元素的值都是 null。

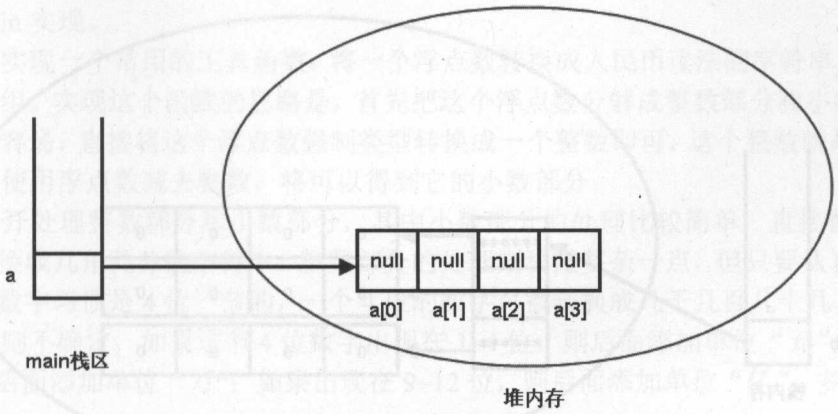


图 5.2 将二维数组当成一维数组初始化的存储示意图

由于 a 数组的元素必须是 Array<Int>数组，所以接下来的程序对 a[0]元素执行初始化，也就是让图 5.2 右边堆内存中的第一个数组元素指向一个有效的数组内存，即指向一个长度为 2 的 Array<Int>数组。

程序执行 a[0] = arrayOf(2, 5)将数组 a 的第一个元素设为长度为 2 的数组，该数组的第一个元素是 2，第二个元素是 5。

然后程序显式为 a[0]数组的第二个元素赋值为 6。此时在内存中的存储示意图如图 5.3 所示。

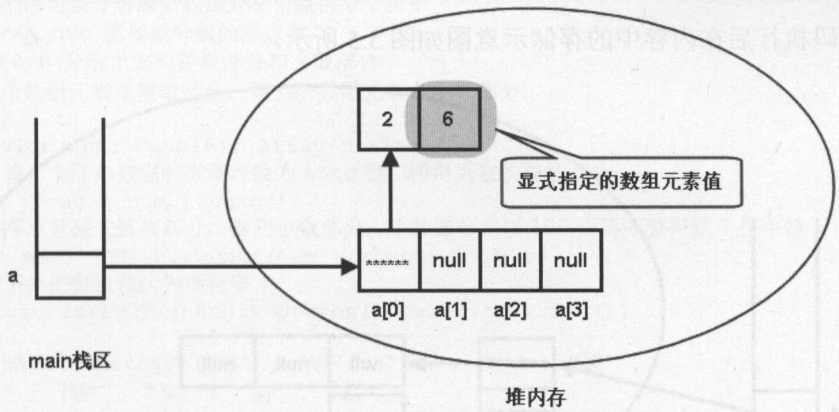


图 5.3 初始化 a[0]后的存储示意图

图 5.3 中灰色区域的数组元素就是程序显式指定的数组元素值。接下来程序迭代输出 a[0]数组的每个数组元素，将看到输出 2 和 6。

可能有读者会怀念以前 Java 数组同时初始化两个维数的用法，Kotlin 同样可以提供支持。例如下面代码（程序清单同上）：

```
// 同时初始化二维数组的两个维数
var b = Array<IntArray>(3, {IntArray(4, {0})})
```

上面代码将定义一个 b 数组变量，这个数组变量指向一个长度为 3 的数组，该数组的每个数组元素又是一个数组类型，它们各指向对应的长度为 4 的 IntArray 数组，每个数组元素的值为 0。这行代码执行后在内存中的存储示意图如图 5.4 所示。

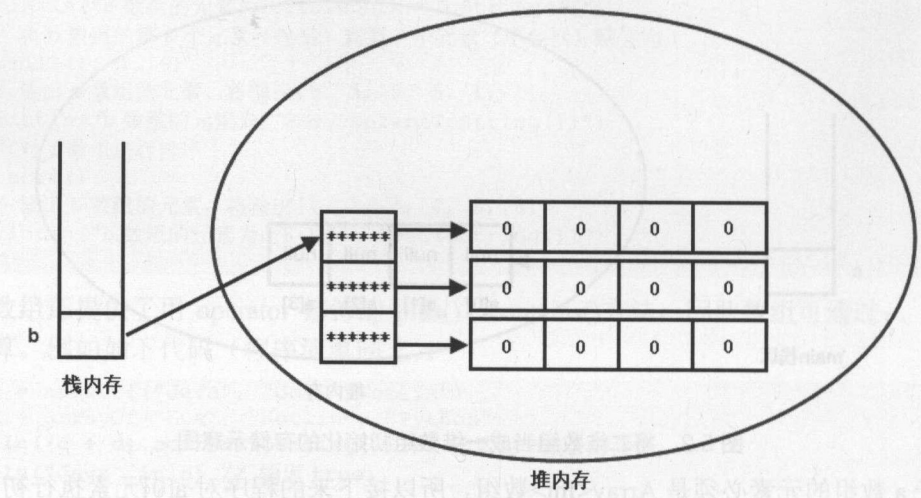


图 5.4 同时初始化二维数组的两个维数后的存储示意图

也可以使用 arrayOf()方法来初始化二维数组。使用 arrayOf()方法来初始化二维数组时，二维数组的每个数组元素都是一维数组，因此必须指定多个一维数组作为二维数组的初始化值。例如下面代码（程序清单同上）：

```
// 使用 arrayOf 来初始化一个二维数组
var str1 =arrayOf(arrayOfNulls<String>(3)
    , arrayOf("hello"))
```

上面代码执行后在内存中的存储示意图如图 5.5 所示。

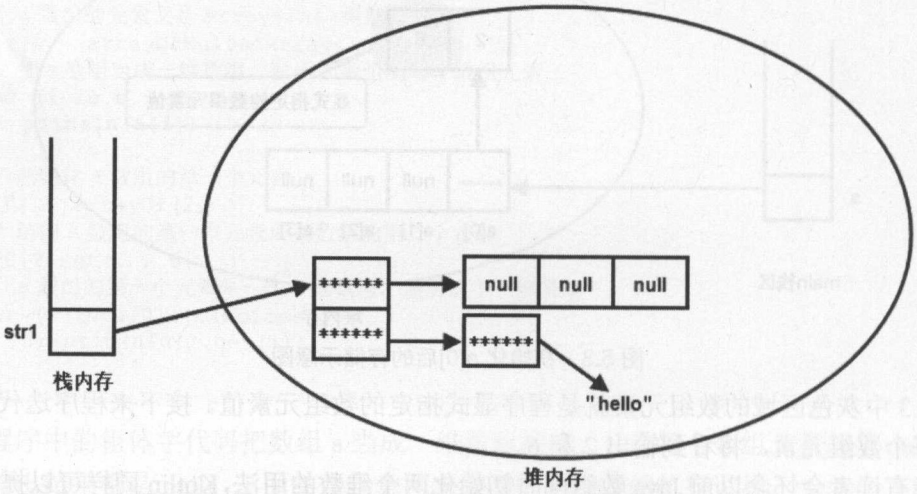


图 5.5 采用静态初始化语法初始化二维数组的存储示意图

通过上面讲解可以得到一个结论：二维数组是一维数组，其数组元素是一维数组；三维数组也是一维数组，其数组元素是二维数组；四维数组还是一维数组，其数组元素是三维数组……因此，只要按照这个思路来做，Kotlin 完全可以支持多维数组。

5.1.7 数组的应用举例

数组的用途很广泛，本节将会把《疯狂 Java 讲义》一书中关于数组应用的两个示例转换

为使用 Kotlin 实现。

下面将实现一个常用的工具函数：将一个浮点数转换成人民币读法的字符串，这个程序就需要使用数组。实现这个函数的思路是，首先把这个浮点数分解成整数部分和小数部分。提取整数部分很容易，直接将这个浮点数强制类型转换成一个整数即可，这个整数就是浮点数的整数部分；再使用浮点数减去整数，将可以得到它的小数部分。

然后分开处理整数部分和小数部分，其中小数部分的处理比较简单，直接截断到保留 2 位数字，转换成几角几分的字符串。整数部分的处理则稍微复杂一点，但只要认真分析不难发现，中国的数字习惯是 4 位一节的，一个 4 位的数字可被转换成几千几百几十几，至于后面添加什么单位则不确定，如果这节 4 位数字出现在 1~4 位，则后面添加单位“元”；如果出现在 5~8 位，则后面添加单位“万”；如果出现在 9~12 位，则后面添加单位“亿”；多于 12 位就暂不考虑了。

因此，实现这个程序的关键就是把一个 4 位的数字字符串转换成汉字字符串。下面程序把这个需求实现了一部分。

程序清单：codes\05\5.1\Num2Rmb.kt

```
fun main(args: Array<String>) {
    // 测试把一个浮点数分解成整数部分和小数部分
    println(divide(236711125.123).contentToString())
    // 测试把一个 4 位的数字字符串变成汉字字符串
    println(toHanStr("6109"))
}
/**
 * 把一个浮点数分解成整数部分和小数部分字符串
 * @param num 需要被分解的浮点数
 * @return 分解出来的整数部分和小数部分
 * 第一个数组元素是整数部分，第二个数组元素是小数部分
 */
fun divide(num: Double): Array<String>{
    // 将一个浮点数强制类型转换为 Long 型，即得到它的整数部分
    var zheng = num.toLong()
    // 浮点数减去整数部分，得到小数部分，小数部分乘以 100 后再取整得到 2 位小数
    var xiao = Math.round((num - zheng) * 100)
    // 下面把整数转换为字符串
    return arrayOf(zheng.toString(), xiao.toString())
}
val hanArr = arrayOf("零" , "壹" , "贰" , "叁" , "肆" ,
    "伍" , "陆" , "柒" , "捌" , "玖")
val unitArr = arrayOf("十" , "百" , "千")
/**
 * 把一个 4 位的数字字符串变成汉字字符串
 * @param numStr 需要被转换的 4 位的数字字符串
 * @return 4 位的数字字符串被转换成汉字字符串
 */
fun toHanStr(numStr: String): String {
    var result = ""
    val numLen = numStr.length
    // 依次遍历数字字符串的每一位数字
    for (i in 0 until numLen ) {
        // 把 Char 型数字转换成 Int 型数字，因为它们的 ASCII 码值恰好相差 48
        // 因此把 Char 型数字减去 48 得到 Int 型数字，例如'4'被转换成 4
        var num = numStr[i].toInt() - 48
        // 如果不是最后一位数字，而且数字不是 0，则需要添加单位（千、百、十）
        if ( i != numLen - 1 && num != 0)
        {
```

```
            result += hanArr[num] + unitArr[numLen - 2 - i]
        }
        // 否则不要添加单位
        else
        {
            result += hanArr[num]
        }
    }
    return result
}
```

运行上面程序，将看到如下运行结果：

```
[236711125, 12]
陆仟壹佰零玖
```

从上面程序的运行结果来看，初步实现了所需功能，但这个程序并不是这么简单的，对0的处理比较复杂。例如，有两个0连在一起时该如何处理呢？如果最高位是0如何处理呢？最低位是0又如何处理呢？因此，这个程序还需要继续完善，希望读者能把这个程序写完。

除此之外，还可以利用二维数组来完成五子棋、连连看、俄罗斯方块、扫雷等常见小游戏。下面简单介绍利用二维数组实现五子棋。先定义一个二维数组作为下棋的棋盘，每当一个棋手下一步棋后，也就是为二维数组的一个数组元素赋值。下面程序完成了这个程序的初步功能。

程序清单：codes\05\5.1\Gobang.kt

```
// 定义棋盘的大小
val BOARD_SIZE = 15
// 定义一个二维数组来充当棋盘
var board = Array(BOARD_SIZE, {Array(BOARD_SIZE, {""})})
fun initBoard(){
    // 把每个元素赋为"╋"，用于在控制台画出棋盘
    for (i in 0 until BOARD_SIZE){
        for (j in 0 until BOARD_SIZE){
            board[i][j] = "╋"
        }
    }
}
// 在控制台输出棋盘的方法
fun printBoard(){
    // 打印每个数组元素
    for (i in 0 until BOARD_SIZE){
        for ( j in 0 until BOARD_SIZE ){
            // 打印数组元素后不换行
            print(board[i][j])
        }
        // 每打印完一行数组元素后输出一个换行符
        println()
    }
}
fun main(args: Array<String>){
    initBoard()
    printBoard()
    var inputStr = readLine()
    while (inputStr != null){
        // 将用户输入的字符串以逗号（,）作为分隔符，分隔成两个字符串
        var posStrArr = inputStr.split(",")
        // 将两个字符串转换成用户下棋的坐标
        var xPos = posStrArr[0].toInt()
        var yPos = posStrArr[1].toInt()
        // 把对应的数组元素赋为"●"
        board[yPos - 1][xPos - 1] = "●"
```

```
        /*
         电脑随机生成两个整数，作为电脑下棋的坐标，赋给 board 数组
         还涉及
            1. 坐标的有效性，只能是数字，不能超出棋盘范围
            2. 下的棋的点，不能重复下棋
            3. 每次下棋后，需要扫描谁赢了了
         */
        printBoard()
        println("请输入您下棋的坐标，应以 x,y 的格式：")
        inputStr = readLine()
    }
}
```

运行上面程序，将看到如图 5.6 所示的界面。

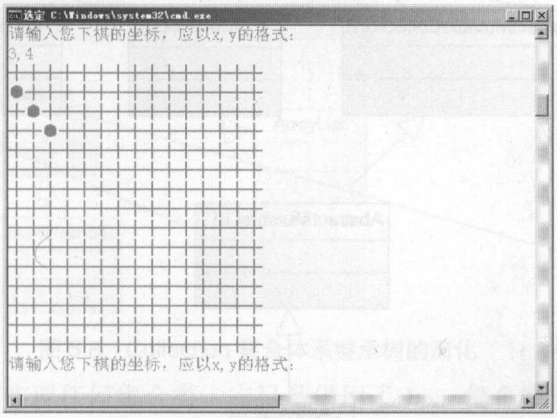


图 5.6 控制台五子棋的运行界面

从图 5.6 来看，程序上面显示的黑点一直是棋手下的棋，电脑还没有下棋，电脑下棋可以通过随机生成两个坐标值来控制，当然也可以增加人工智能（但这已经超出了本书的范围，实际上也很简单）来控制下棋。

除此之外，读者还需要在这个程序的基础上进行完善，保证用户和电脑下的棋的坐标上不能已经有棋子（通过判断对应数组元素只能是"+"来确定），还需要进行 4 次循环扫描，判断横、竖、左斜、右斜是否有 5 个棋子连在一起，从而判定胜负。

5.2 Kotlin 集合概述

Kotlin 的集合类同样由两个接口派生：Collection 和 Map。Collection 和 Map 是 Java 集合框架的根接口，这两个接口又包含了一些子接口或实现类。

Kotlin 集合与 Java 集合不同，Java 集合都是可变集合——开发者可以向集合中添加、删除、修改元素，但 Kotlin 的集合被分成两大类：可变集合和不可变集合。只有可变集合才能添加、删除、修改元素，不可变集合只能读取元素。

如图 5.7 所示是 Collection 接口、子接口及其实现类的继承树。

图 5.7 看上去让人有些头晕，这是因为 Kotlin 为 Collection 接口派生了一个子接口，即 MutableCollection，该子接口又包含了两个子接口：MutableSet 和 MutableList，这一分支就代表了 Kotlin 的可变集合。而 Collection 直接派生的 Set、List 接口则代表了不可变集合。因此，Kotlin 的 Collection 集合体系由 6 个接口组成，其中 3 个是不可变的集合接口；另外 3 个子接口代表了可变集合。

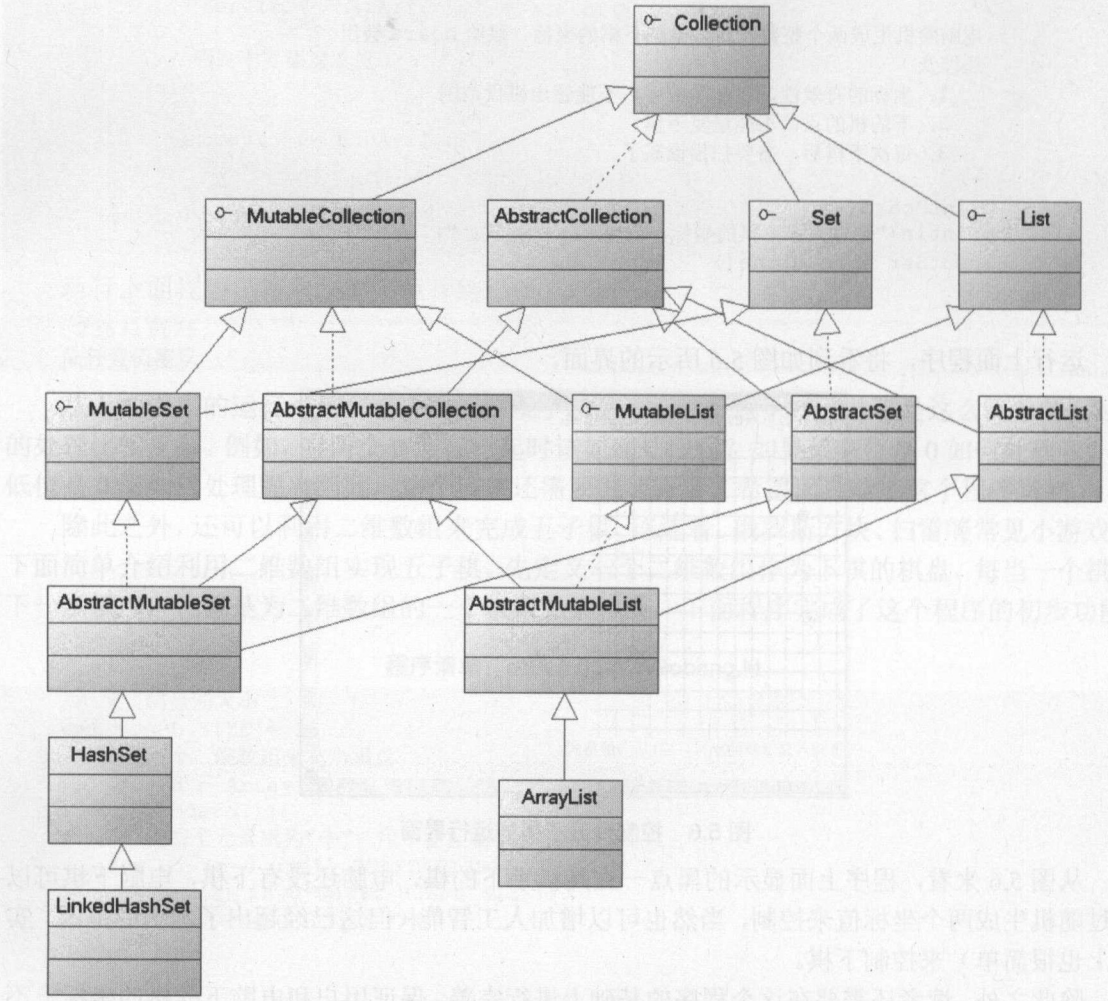


图 5.7 Collection 集合体系的继承树

图 5.7 之所以让人有些头晕，其实是因为我们保留了 Kotlin 集合体系的抽象类。从图 5.7 可以看出，Kotlin 的设计思路有两点：

- 可变集合接口、可变集合类是不可变集合接口、不可变类的子接口或子类。
- 每个接口下都有一个抽象实现类。

如果将图5.7所示继承树中的抽象类去除，此时就只剩下6个集合接口和3个集合实现类，可以看到如图 5.8 所示的继承树。

从图5.8可以看出，Kotlin的集合实现类比Java更少，它只提供了HashSet、LinkedHashSet、ArrayList 这三个最常见的实现类，如果开发者需要使用 TreeSet、LinkedList 集合实现类，则依然可以使用 Java 集合框架提供的类。

注意：

Kotlin 在 JVM 平台上并未真正实现 HashSet、LinkedHashSet、ArrayList 这些类，只是为 Java 的 HashSet、LinkedHashSet、ArrayList 指定了一个类型别名而已，这样就直接借用了 Java 的这些集合类。此外，Kotlin 为这些集合类扩展了更多的方法，因此用起来更方便。

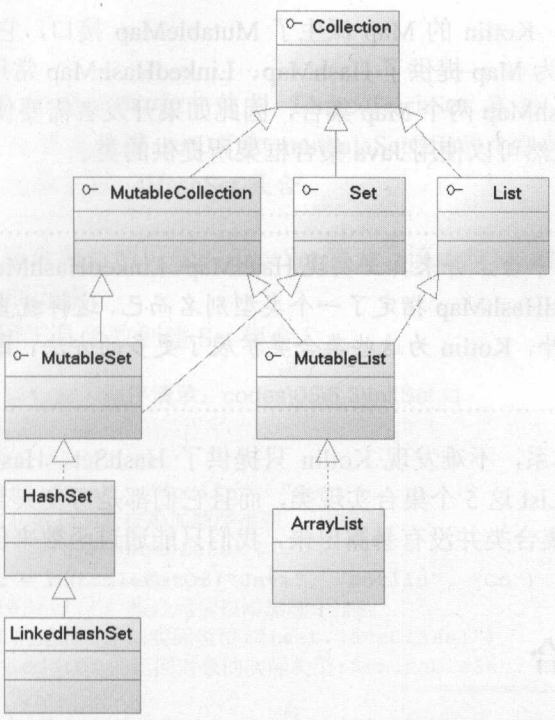


图 5.8 Collection 集合体系继承树的简化

由于 Kotlin 并未真正实现任何集合类，它只是借用了 Java 集合框架原有的集合类，因此本章没必要从底层实现去分析这些集合类的性能差异。如果读者希望了解各集合实现类的性能差异，则可参考《疯狂 Java 讲义》。

Kotlin 也提供了 Map 集合。如图 5.9 所示是 Kotlin 所提供的 Map 集合体系的继承树。

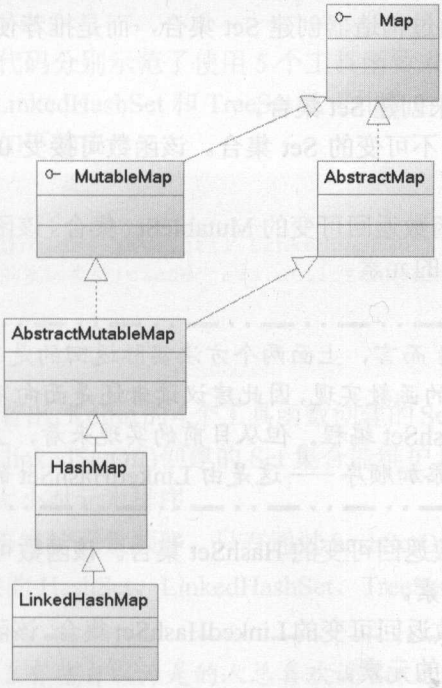


图 5.9 Kotlin 提供的 Map 集合体系的继承树

从图 5.9 可以看出，Kotlin 的 Map 派生了 MutableMap 接口，它代表了一个可变的 Map 集合。而实际上 Kotlin 为 Map 提供了 HashMap、LinkedHashMap 常用实现类。Kotlin 只提供了 HashMap、LinkedHashMap 两个 Map 集合，因此如果开发者需要使用 TreeMap、EnumMap 或 WeakHashMap，则依然可以使用 Java 集合框架所提供的类。

注意：

Kotlin 在 JVM 平台上并未真正实现 HashMap、LinkedHashMap，只是为 Java 的 HashMap、LinkedHashMap 指定了一个类型别名而已，这样就直接借用了 Java 的这些集合类。此外，Kotlin 为这些集合类扩展了更多的方法，因此用起来更方便。



纵观 Kotlin 集合体系，不难发现 Kotlin 只提供了 HashSet、HashMap、LinkedHashSet、LinkedHashMap、ArrayList 这 5 个集合实现类，而且它们都是可变集合。那么说好的不可变集合呢？Kotlin 的不可变集合类并没有暴露出来，我们只能通过函数来创建不可变集合，在本章后面会见到相关内容。

5.3 Set 集合

与 Java 集合相似，Kotlin 的 Collection 集合和 Set 集合的功能基本相同，Set 集合只是为 Collection 集合增加了额外的限制：集合元素不允许重复。除此之外，Set 集合与 Collection 集合提供的方法大致相同。

5.3.1 声明和创建 Set 集合

实际上 Kotlin 并没有真正为 JVM 平台实现任何 Set 集合类（只是通过别名借用了 Java 集合框架的类），因此不推荐通过构造器创建 Set 集合，而是推荐使用 Kotlin 提供的工具函数来创建 Set 集合。

Kotlin 提供了如下函数来创建 Set 集合。

- setOf()：该函数返回不可变的 Set 集合。该函数可接受 0 个或多个参数，这些参数将作为集合的元素。
- mutableSetOf()：该函数返回可变的 MutableSet 集合。该函数可接受 0 个或多个参数，这些参数将作为集合的元素。



提示：

对于 Kotlin 1.1 而言，上面两个方法实际返回的是 LinkedHashSet 的实例，但这取决于是 Kotlin 的函数实现，因此建议读者还是面向不可变 Set、可变 Set 编程，不要面向 LinkedHashSet 编程。但从目前的实现来看，上面两个方法返回的 Set 集合都会记住元素的添加顺序——这是由 LinkedHashSet 的特征所决定的。

- hashSetOf()：该函数返回可变的 HashSet 集合。该函数可接受 0 个或多个参数，这些参数将作为集合的元素。
- linkedSetOf()：该函数返回可变的 LinkedHashSet 集合。该函数可接受 0 个或多个参数，这些参数将作为集合的元素。

注意：

如果读者明确希望在程序中使用 LinkedHashSet 集合，则建议使用 linkedSetOf()函数，而不要使用 setOf()和 mutbaleSetOf()两个函数，虽然它们在 Kotlin 1.1 中确实也返回 LinkedHashSet 集合。



- sortedSetOf()：该函数返回可变的 TreeSet 集合。该函数可接受 0 个或多个参数，这些参数将作为集合的元素。

如下程序示范了使用工具函数创建 Set 集合。

程序清单：codes\05\5.3\InitSet.kt

```
fun main(args: Array<String>) {
    // 创建不可变集合，返回值是 Set
    var set = setOf("Java", "Kotlin", "Go")
    println(set) // 集合元素按添加顺序排列
    // 创建可变集合，返回值是 MutableSet
    var mutableSet = mutableSetOf("Java", "Kotlin", "Go")
    println(mutableSet) // 集合元素按添加顺序排列
    println("setOf 的返回对象的实际类型:${set.javaClass}")
    println("mutableSetOf 的返回对象的实际类型:${mutableSet.javaClass}")
    // 创建 HashSet 集合
    var hashSet = hashSetOf("Java", "Kotlin", "Go")
    println(hashSet) // 不保证元素的顺序
    // 创建 LinkedHashSet 集合
    var linkedHashSet = linkedSetOf("Java", "Kotlin", "Go")
    println(linkedHashSet) // 集合元素按添加顺序排列
    // 创建 TreeSet 集合
    var treeSet = sortedSetOf("Java", "Kotlin", "Go")
    println(treeSet) // 集合元素由小到大排列
}
```

上面程序中 5 行粗体字代码分别示范了使用 5 个工具函数来创建 Set 集合，这些函数完全可以创建 Java 的 HashSet、LinkedHashSet 和 TreeSet 这三个集合类。

运行上面程序，可看到如下输出：

```
[Java, Kotlin, Go]
[Java, Kotlin, Go]
setOf 的返回对象的实际类型:class java.util.LinkedHashSet
mutableSetOf 的返回对象的实际类型:class java.util.LinkedHashSet
[Go, Java, Kotlin]
[Java, Kotlin, Go]
[Go, Java, Kotlin]
```

从上面的运行结果可以看出，Kotlin 的 4 个工具函数创建的 Set 集合都能维护元素的顺序，其中 setOf()、mutableSetOf()、linkedSetOf()创建的 Set 集合能维护元素的添加顺序，sortedSetOf()函数创建的 Set 集合会按照大小对元素排序。

如果真正希望 Set 集合不维护元素顺序，只有通过 hashSetOf()函数创建 Set 集合才行。这依然是由 Java 集合框架提供的 HashSet、LinkedHashSet、TreeSet 实现类的特征所决定的。



提示：

经常看到网络上有些自以为是的人总喜欢说 Set 是无序集合（还有不少面试者喜欢考这个题目）。实际上，Set 只是一个接口，而该接口下常用的 3 个实现类

> HashSet、LinkedHashSet 和 TreeSet 中有两个是有序的。因此，笼统地说 Set 集合是无序集合的人，基本都是不懂装懂。

Set 集合创建完成后，就可以使用 Set 集合了，包括获取 Set 集合中元素的个数、判断是否包含某个元素等。

5.3.2 使用 Set 的方法

与前面介绍的 Array 类相似，除 Java 原生 Set 的各种方法之外，Kotlin 的 Set 还扩展了大量方法，在 Set 集合中有很多方法与 Array 的方法功能相似，例如 all（判断是否所有集合元素都满足指定规则）、any（判断是否任一元素都满足指定规则）、associate（根据 Set 集合生成 Map 集合）、drop（按规则去除集合的部分元素）、filter（对集合元素执行过滤）、find（查找集合元素）……前面已经大致介绍过这些方法，此处不再详述。下面通过代码来示范 Set 集合的功能。

程序清单：codes\05\5.3\SetFun.kt

```
fun main(args: Array<String>) {
    // 创建不可变集合，返回值是 Set
    var set = setOf("Java", "Kotlin", "Go")
    // 判断是否所有元素的长度都大于 4
    println(set.all({it.length > 4})) // 输出 false
    // 判断是否任一元素的长度都大于 4
    println(set.any({it.length > 4})) // 输出 true
    // 以 Lambda 表达式的值为 key，集合元素为 value，组成 Map 集合
    val map = set.associateBy({"《疯狂" + it + "讲义》"})
    // 输出：{《疯狂 Java 讲义》=Java, 《疯狂 Kotlin 讲义》=Kotlin, 《疯狂 Go 讲义》=Go}
    println(map)
    // 由于有 contains()方法，所以可用 in、!in 运算符
    println("Java" in set) // 输出 true
    println("Go" !in set) // 输出 false
    // 返回删除 Set 集合前面两个元素后的集合
    val dropedList = set.drop(2)
    println(dropedList) // 输出[Go]
    // 对 Set 集合元素进行过滤：要求集合元素包含 li
    val filteredList = set.filter({"li" in it})
    println(filteredList) // 输出[Kotlin]
    // 查找 Set 集合中包含 li 的元素，如果找到就返回该元素，否则返回 null
    val foundStr1 = set.find({"li" in it})
    println(foundStr1) // 输出 Kotlin
    // 查找 Set 集合中包含 gang 的元素，如果找到就返回该元素，否则返回 null
    val foundStr2 = set.find({"gang" in it})
    println(foundStr2) // 输出 null
    // 将 Set 集合中的所有字符串拼接在一起
    val foldedList = set.fold("", {acc, e -> acc + e})
    println(foldedList) // 输出 JavaKotlinGo
    // 查找某个元素的出现位置
    println(set.indexOf("Go")) // 输出 2
    // 将每个集合元素映射成新值，返回所有新值组成的 Set 集合
    val mappedList = set.map({"《疯狂" + it + "讲义》"})
    // 输出：[《疯狂 Java 讲义》, 《疯狂 Kotlin 讲义》, 《疯狂 Go 讲义》]
    println(mappedList)
    // 获取最大值
    println(set.max()) // 输出 Kotlin
    // 获取最小值
```

```
    println(set.min()) // 输出 Go
    // 反转集合顺序
    val reversedList = set.reversed()
    println(reversedList) // 输出[Go, Kotlin, Java]
    var bSet = setOf("Lua", "Erlang", "Kotlin")
    // 计算两个集合的交集
    println(set intersect bSet) // 输出[Kotlin]
    // 计算两个集合的并集
    println(set union bSet) // 输出[Java, Kotlin, Go, Lua, Erlang]
    // 由于 Set 有 operator 修饰的 plus、minus 方法，因此可执行+、-运算
    // 集合相加，相当于并集
    println(set + bSet) // 输出[Java, Kotlin, Go, Lua, Erlang]
    // 集合相减，减去它们公共的元素
    println(set - bSet) // 输出[Java, Go]
}
```

上面程序对 Set 集合的大部分方法都进行了示范，读者可根据这些方法来体验 Kotlin 的 Set 集合的方便之处。

上面程序中后两行粗体字代码使用了+、-运算符对集合进行操作。这是由于 Set 集合提供了 operator 修饰的 plus、minus 方法，因此可使用+、-运算符操作集合。

上面程序中前两行粗体字代码使用了 intersect、union 运算符来操作 Set 集合，这也是由 Kotlin 语言的特征所决定的——Kotlin 规定以 infix 修饰的方法，能以运算符的方式进行调用。

5.3.3 遍历 Set

Kotlin 也支持使用 for-in 循环遍历 Set，与遍历数组的方式基本相同。

下面程序示范了如何使用 for-in 循环来遍历 Set 集合的元素。

程序清单：codes\05\5.3\ForinTest.kt

```
fun main(args: Array<String>) {
    var books = setOf("疯狂 Android 讲义" ,
        "疯狂 iOS 讲义",
        "疯狂 Kotlin 讲义")
    // 使用 for-in 循环来遍历 Set
    // 其中 book 将会自动迭代 Set 集合的每个元素
    for( book in books) {
        println(book)
    }
}
```

从上面程序中可以看出，使用 for-in 循环遍历 Set 元素时无须获得 Set 元素的个数，也无须根据索引来访问 Set 元素，for-in 循环自动迭代 Set 的每个元素，当每个元素都被迭代一次后，for-in 循环自动结束。

与遍历、访问数组元素时类似，使用 for-in 循环遍历 Set 集合时也不允许对循环常量进行赋值——这是因为 for-in 循环隐式使用 let 来声明该常量。

此外，Set 集合继承了 Iteratable，因此可使用该接口中定义的 forEach()方法来遍历集合。该方法需要接受一个 Lambda 表达式作为参数，该 Lambda 表达式负责对集合元素进行处理。例如如下代码（程序清单同上）：

```
// 调用 forEach 方法来遍历 Set 集合
books.forEach({println(it)})
```

由于 setOf()方法返回的 Set 集合是有序的，是有序的（重要的事情再说一次）！因此程序可以通过索引来遍历 Set 集合，Set 集合提供了 indices 方法返回其索引的区间。例如如下代码

（程序清单同上）：

```
// 根据索引来遍历 Set 集合
for (i in books.indices) {
    println(books.elementAt(i))
}
```

5.3.4 可变的 Set

除使用 setOf()函数返回的集合是不可变集合之外，使用 mutableSetOf()、hashSetOf()、linkedSetOf()、sortedSetOf()函数返回的集合都是可变的，其中后面三个函数返回的集合类型都是明确的，依次是 HashSet、LinkedHashSet、TreeSet。

创建可变的 Set 集合之后，接下来就可以对该 Set 的元素执行添加、删除操作了。

1. 添加元素

Set 提供了 add(element: E)方法来添加元素，每调用该方法一次，程序就会向 Set 中添加一个元素，Set 的长度也会自动加 1。Set 也提供了 addAll(elements: Collection<E>)方法来批量添加多个元素。

如下程序调用了 Set 的 add(element: E)方法来添加元素。

程序清单：codes\05\5.3\Insert.kt

```
fun main(args: Array<String>) {
    // 定义一个可变的 Set
    var languages = mutableSetOf("Swift")
    // 添加一个元素
    languages.add("Go")
    languages.add("Lua")
    // ["Lua", "Go", "Swift"]
    println(languages)
    println(languages.count()) // 输出 3
    languages.addAll(setOf("Java", "Kotlin"))
    // [Swift, Go, Lua, Java, Kotlin]
    println(languages)
}
```

上面程序中的三行粗体字代码分别示范了向可变的 Set 集合中添加元素的方法。

2. 删除元素

Set 提供了如下方法来删除元素。

- remove(element: E)：删除指定元素，删除成功则返回 true。
- removeAll(elements: Collection<E>)：批量删除 Set 集合中的多个元素。
- retainAll(elements: Collection<E>)：只保留 Set 集合中与 elements 集合共有的元素。
- clear()：清空集合。

如下程序调用了上面方法来删除 Set 元素。

程序清单：codes\05\5.3\Remove.kt

```
fun main(args: Array<String>) {
    var languages = mutableSetOf("Kotlin", "OC", "PHP", "Perl" , "Ruby" , "Go")
    // 删除"PHP"
    languages.remove("PHP")
    // 再次删除"Perl"
    languages.remove("Perl")
    println(languages)  // 输出[Kotlin, OC, Ruby, Go]
    // 批量删除多个元素
```

```
    languages.removeAll(setOf("Ruby", "Go"))
    println(languages)  // 输出[Kotlin, OC]
    // 清空 Set 集合
    languages.clear()
    println(languages.count())  // 输出 0
}
```

上面程序中的粗体字代码示范了 Set 集合删除元素的方法。

此外，Set 和 MutableSet 都包含一个 iterator()方法，但普通 Set 的 iterator ()方法返回的是 Iterator 对象，该 Iterator 对象只有 hasNext()和 next()两个方法；而 MutableSet 的 iterator()方法返回的是 MutableIterator 对象，该对象除 hasNext()和 next()两个方法之外，还提供了一个 remove()方法，该方法可用于在遍历时删除元素。



提示：

简单来说，Kotlin 的 MutableIterator 才相当于 Java 的 Iterator。

例如如下代码（程序清单同上）：

```
var set = mutableSetOf("aa", "eyz" , "abc")
var it = set.iterator() // 返回 MutableIterator
while(it.hasNext()){
    var e = it.next()
    println(e)
    // 遍历时删除元素
    if(e.length < 3) {
        it.remove()
    }
}
println(set) // 输出["eyz" , "abc"]
```

5.4 List 集合

与 Java 类似，List 集合的最大特征就是集合元素都有对应的顺序索引。List 集合允许使用重复元素，可以通过索引来访问指定位置的集合元素。List 集合默认按元素的添加顺序设置元素的索引，例如第一次添加的元素索引为 0，第二次添加的元素索引为 1……

5.4.1 声明和创建 List 集合

正如前面所介绍的，Kotlin 同样并未真正实现 List 集合，它只是通过别名借用了 Java 体系中的 ArrayList 集合。因此不推荐通过构造器来创建 List 集合，而是推荐使用工具函数来创建 List 集合。Kotlin 提供了如下工具函数来创建 List 集合。

- listOf()：该函数返回不可变的 List 集合。该函数可接受 0 个或多个参数，这些参数将作为集合的元素。
- listOfNotNull()：该函数返回不可变的 List 集合。该函数与前一个函数的唯一区别是，该函数会自动去掉传入的一系列参数中的 null 值。简而言之，该函数返回的 List 集合不包含 null 值。
- mutableListOf()：该函数返回可变的 MutableList 集合。该函数可接受 0 个或多个参数，这些参数将作为集合的元素。



提示：

对于 Kotlin 1.1 而言，上面的第一个函数实际返回的是 java.util.Arrays$ArrayList 的实例，这依然是 JDK 所提供的不可变的 List 集合。后面两个函数实际返回的是大家无比熟悉的 ArrayList。

- arrayListOf()：该函数返回可变的 ArrayList 集合。该函数可接受 0 个或多个参数，这些参数将作为集合的元素。

注意：

如果读者明确希望在程序中使用 ArrayList 集合，则建议使用 arrayListOf()函数，而不要使用 mutableListOf()函数，虽然它在 Kotlin 1.1 中确实也返回 ArrayList 集合。



如下程序示范了使用工具函数创建 List 集合。

程序清单：codes\05\5.4\InitList.kt

```
fun main(args: Array<String>) {
    // 创建不可变集合，返回值是 List
    var list1 = listOf("Java", "Kotlin", null, "Go")
    println(list1) // 集合包含 null 值
    // 创建不可变集合，返回值是 List
    var list2 = listOfNotNull("Java", "Kotlin", null, "Go")
    println(list2) // 集合不包含 null 值
    // 创建可变集合，返回值是 MutableList
    var mutableList = mutableListOf("Java", "Kotlin", null, "Go")
    println(mutableList)
    println("listOf 的返回对象的实际类型:${list1.javaClass}")
    println("listOfNotNull 的返回对象的实际类型:${list2.javaClass}")
    println("mutableList 的返回对象的实际类型:${mutableList.javaClass}")
    // 创建 ArrayList 集合
    var arrayList = arrayListOf("Java", "Kotlin", null, "Go")
    println(arrayList)
}
```

上面程序中的 4 行粗体字代码分别示范了使用 4 个工具函数来创建 List 集合，这 4 个工具函数可用于创建 Kotlin 的不可变 List 和可变 List 集合。

运行上面程序，可看到如下输出：

```
[Java, Kotlin, null, Go]
[Java, Kotlin, Go]
[Java, Kotlin, null, Go]
listOf 的返回对象的实际类型:class java.util.Arrays$ArrayList
listOfNotNull 的返回对象的实际类型:class java.util.ArrayList
mutableList 的返回对象的实际类型:class java.util.ArrayList
[Java, Kotlin, null, Go]
```

5.4.2 使用 List 的方法

List 同样提供了与 Set 相似的集合操作方法。通常来说，Set 支持的操作，List 一般都能支持，它还增加了通过索引操作集合元素的方法。

- get：带 operator 修饰的方法，因此可用“[]“运算符访问集合元素。
- indexOf：返回集合元素在 List 中的索引。

- lastIndexOf：返回集合元素在 List 中最后一次的出现位置。
- subList：返回 List 集合的子集合。

如下程序示范了 List 集合额外增加的通过索引访问集合元素的方法。

程序清单：codes\05\5.4\ListFun.kt

```
fun main(args: Array<String>) {
    // 创建不可变集合，返回值是 List
    var list = listOf("Java", "Kotlin", null, "Go")
    for (i in list.indices ) {
        // 使用[]运算符访问集合元素，实际上就是根据 get 方法访问
        println(list[i])
    }
    // 获取指定元素的出现位置
    println(list.indexOf("Kotlin")) // 输出 1
    // 获取 List 的子集合
    println(list.subList(1, 3)) // 输出["Kotlin", null]
}
```

5.4.3 可变的 List

使用 mutableListOf()、arrayListof()函数返回的 List 集合都是可变的，其中后面一个函数返回的集合类型都是明确的 ArrayList。

创建可变的 List 集合之后，接下来就可以对该 List 的元素执行添加、插入、删除、替换等操作了。原来在可变的 Set 中介绍的方法，可变的 List 也完全支持，它还增加了一些根据索引执行插入、删除、替换的方法。

如下程序示范了对可变 List 执行插入、删除、替换操作。

程序清单：codes\05\5.4\MutableListTest.kt

```
fun main(args: Array<String>) {
    // 创建可变集合，返回值是 List
    var mutableList = mutableListOf("Kotlin", null, "Go")
    // 在索引 2 处插入一个新元素
    mutableList.add(2, "Java")
    println(mutableList) // [Kotlin, null, Java, Go]
    // 删除索引 1 处的元素
    mutableList.removeAt(1)
    println(mutableList) // [Kotlin, Java, Go]
    // 将索引 1 处的元素替换为"Lua"
    mutableList[1] = "Lua"
    println(mutableList) // [Kotlin, Lua, Go]
    // 清空 List 集合的所有元素
    mutableList.clear()
    println(mutableList.size) // 0
}
```

5.5 Map 集合

与 Java 相同的是，Kotlin 的 Map 集合同样用于保存 key-value 对；与 Java 不同的是，Kotlin 的 Map 集合也被分为可变的和不可变的。

5.5.1 声明和创建 Map 集合

实际上 Kotlin 并没有真正为 JVM 平台实现任何 Map 集合类（只是通过别名借用了 Java

集合框架的类），因此不推荐通过构造器创建 Map 集合，而是推荐使用 Kotlin 提供的工具函数来创建 Map 集合。

Kotlin 提供了如下函数来创建 Map 集合。

- mapOf()：该函数返回不可变的 Map 集合。该函数可接受 0 个或多个 key-value 对，这些 key-value 对将作为 Map 的元素。
- mutableMapOf()：该函数返回可变的 MutableMap 集合。该函数可接受 0 个或多个 key-value 对，这些 key-value 对将作为集合的元素。



提示：

对于 Kotlin 1.1 而言，上面两个函数实际返回的是 LinkedHashMap 的实例，但这取决于是 Kotlin 的函数实现，因此建议读者还是面向不可变的 Map、可变的 Map 编程，不要面向 LinkedHashMap 编程。但从目前的实现来看，上面两个方法返回的 Map 集合都会记住 key-value 对的添加顺序——这是由 LinkedHashMap 的特征所决定的。

- hashMapOf()：该函数返回可变的 HashMap 集合。该函数可接受 0 个或多个 key-value 对，这些 key-value 对将作为 Map 的元素。
- linkedMapOf()：该函数返回可变的 LinkedHashMap 集合。该函数可接受 0 个或多个 key-value 对，这些 key-value 对将作为 Map 的元素。



注意：

如果读者明确希望在程序中使用 LinkedHashMap 集合，则建议使用 linkedMapOf()函数，而不要使用 mapOf()、mutbaleMapOf()这两个函数，虽然它们在 Kotlin 1.1 中确实也返回 LinkedHashMap 集合。



- sortedMapOf()：该函数返回可变的 TreeMap 集合。该函数可接受 0 个或多个 key-value 对，这些 key-value 对将作为 Map 的元素。

如下程序示范了使用工具函数创建 Map 集合。

程序清单：codes\05\5.5\InitMap.kt

```
fun main(args: Array<String>) {
    // 创建不可变集合，返回值是 Map
    var map = mapOf("Java" to 86, "Kotlin" to 92, "Go" to 78)
    println(map) // key-value 对按添加顺序排列
    // 创建可变集合，返回值是 MutableMap
    var mutableMap = mutableMapOf("Java" to 86, "Kotlin" to 92, "Go" to 78)
    println(mutableMap) // key-value 对按添加顺序排列
    println("mapOf 的返回对象的实际类型:${map.javaClass}")
    println("mutableMapOf 的返回对象的实际类型:${mutableMap.javaClass}")
    // 创建 HashMap 集合
    var hashMap = hashMapOf("Java" to 86, "Kotlin" to 92, "Go" to 78)
    println(hashMap) // 不保证 key-value 对的顺序
    // 创建 LinkedHashMap 集合
    var linkedHashMap = linkedMapOf("Java" to 86, "Kotlin" to 92, "Go" to 78)
    println(linkedHashMap) // key-value 对按添加顺序排列
    // 创建 TreeMap 集合
    var treeMap = sortedMapOf("Java" to 86, "Kotlin" to 92, "Go" to 78)
    println(treeMap) // key-value 对按 key 由小到大排列
}
```

从上面代码可以看出，Kotlin 需要使用 to 指定 key-value 对，其中 to 之前的是 key，to 之后的是 value。

上面程序中 5 行粗体字代码分别示范了使用 5 个工具函数来创建 Map 集合，这些工具函数完全可以创建 Java 的 HashMap、LinkedHashMap 和 TreeMap 这三个集合类。

运行上面程序，可看到如下输出：

```
{Java=86, Kotlin=92, Go=78}
{Java=86, Kotlin=92, Go=78}
mapOf 的返回对象的实际类型:class java.util.LinkedHashMap
mutableMapOf 的返回对象的实际类型:class java.util.LinkedHashMap
{Go=78, Java=86, Kotlin=92}
{Java=86, Kotlin=92, Go=78}
{Go=78, Java=86, Kotlin=92}
```

从上面的运行结果可以看出，上面有 4 个工具函数创建的 Map 集合都能维护 key-value 对的顺序，其中 mapOf()、mutableMapOf()、linkedMapOf()创建的 Map 集合能维护元素的添加顺序，sortedMapOf()函数创建的 Map 集合会按照 key 大小对 key-value 对排序。

如果真正希望 Map 集合不保证 key-value 对的顺序，只有通过 hashMapOf()函数创建 Map 集合才行。这也是由 Java 集合框架提供的 HashMap、LinkedHashMap、TreeMap 实现类的特征所决定的。



提示：

正如《疯狂 Java 讲义》中所介绍的，Java 的 Set 集合其实是由 Map 集合所维护的，因此如果有某个 Set，那么就一定有对应的 Map。

Map 集合创建完成后，就可以使用 Map 集合了，包括获取 Map 集合中元素的个数、判断是否包含某个元素等。

5.5.2 使用 Map 的方法

Kotlin 为 Map 集合同样扩展了大量方法，这些方法与 Kotlin 为 Set 集合提供的方法也大致相似，区别只是 Map 所提供的方法通常会对 key-value 对整体进行操作，部分方法名包含 key 的方法只对所有 key 进行操作，部分方法名包含 value 的方法则只对所有 value 进行操作。

下面通过程序来示范 Map 集合的功能。

程序清单：codes\05\5.5\MapFun.kt

```
fun main(args: Array<String>) {
    // 创建不可变集合，返回值是 Map
    var map = mapOf("Java" to 86, "Kotlin" to 92, "Go" to 76)
    // 判断是否所有 key-value 对的 key 的长度都大于 4、value 都大于 80
    println(map.all({it.key.length > 4 && it.value > 80})) // 输出 false
    // 判断是否任一 key-value 对的 key 的长度都大于 4、value 都大于 80
    println(map.any({it.key.length > 4 && it.value > 80}))  // 输出 true
    // 由于有 contains()方法，所以可用 in、!in 运算符
    println("Java" in map) // 输出 true
    println("Go" !in map) // 输出 false

    // 对 Map 集合元素进行过滤：要求 key 包含 li
    val filteredMap = map.filter({"li" in it.key})
    println(filteredMap) // 输出{Kotlin=92}
    // 将每个 key-value 对映射成新值，返回所有新值组成的 Map 集合
    val mappedList = map.map({"《疯狂${it.key}讲义》价格为：${it.value}"})
```

```
        // 输出：[《疯狂 Java 讲义》价格为：86，《疯狂 Kotlin 讲义》价格为：92，《疯狂 Go 讲义》价格为：76]
        println(mappedList)
        // 根据 key 获取最大值
        println(map.maxBy({it.key})) // 输出 Kotlin=92
        // 根据 value 获取最小值
        println(map.minBy({it.value})) // 输出 Go=76
        var bMap = mapOf("Lua" to 67, "Erlang" to 73, "Kotlin" to 92)
        // 集合相加，相当于并集
        println(map + bMap) // 输出{Java=86, Kotlin=92, Go=76, Lua=67, Erlang=73}
        // 集合相减，减去它们公共的元素
        println(map - bMap) // 输出{Java=86, Kotlin=92, Go=76}
    }
```

上面程序对 Map 集合的大部分方法都进行了示范，读者可根据这些方法来体验 Kotlin 的 Map 集合的方便之处。

上面程序中两行粗体字代码使用+、-运算符对集合进行操作。这是由于 Map 集合提供了 operator 修饰的 plus、minus 方法，因此可使用+、-运算符操作集合。

5.5.3 遍历 Map

Map 集合由多个 key-value 对组成，因此遍历 Map 集合时既可通过对 key-value 对进行遍历，也可先遍历 key，再通过 key 来获取对应的 value 进行遍历。

Kotlin 的 Map 提供了 operator 修饰的 get()方法，因此可通过“[]”运算符根据 key 来获取 value。

此外，Map 也可直接用 for-in 循环进行遍历，这时循环变量的类型是 Entry（即 key-value 对）。

下面程序示范了对 Map 集合的几种遍历方式。

程序清单：codes\05\5.5\MapIterator.kt

```
fun main(args: Array<String>) {
    // 创建不可变集合，返回值是 Map
    var map = mapOf("Java" to 86, "Kotlin" to 92, "Go" to 76)
    // 遍历 Map 的 key-value 对，entris 元素返回 key-value 对组成的 Set
    for (en in map.entries) {
        println("${en.key} -> ${en.value}")
    }
    // 先遍历 Map 的 key，再通过 key 获取 value
    for (key in map.keys ) {
        println("${key} -> ${map[key]}")
    }
    // 直接用 for-in 循环遍历 Map
    for( (key,value) in map){
        println("${key} -> ${value}")
    }
    // 用 Lambda 表达式遍历 Map
    map.forEach({println("${it.key} -> ${it.value}")})
}
```

上面的粗体字代码一共示范了 4 种遍历 Map 的方式，但这并不是全部，读者只要真正理解了 Map 这种集合，其实完全可根据自己的需要来遍历 Map 集合。

5.5.4 可变的 Map

除使用 mapOf()函数返回的集合是不可变集合之外，使用 mutableMapOf()、hashMapOf()、

linkedMapOf()、sortedMapOf()函数返回的集合都是可变的，其中后面三个函数返回的集合类型都是明确的，依次是 HashMap、LinkedHashMap、TreeMap。

创建可变的 Map 集合之后，接下来就可以对该 Map 的元素执行添加、删除、替换操作了。

可变的 Map 为操作 key-value 对提供了如下方法。

- clear()：清空所有的 key-value 对。
- put(key: K, value: V)：放入 key-value 对。如果原来已有该 key，那么新放入的 value 会覆盖原有的 value。
- putAll(from: Map<out K, V>)：批量放入多个 key-value 对。
- remove(key: K)：删除 key-value 对。

此外，MutableMap 还提供了一个 set(key, value)方法来放入 key-value 对，这意味着程序可通过方括号“[]”运算符来放入 key-value 对。例如如下程序。

程序清单：codes\05\5.5\MutableMapTest.kt

```
fun main(args: Array<String>) {
    var mutableMap = mutableMapOf("OC" to 96, "PHP" to 3400,
        "Perl" to 4300 , "Ruby" to 5600 , "Go" to 5600)
    // 以方括号语法放入 key-value 对
    mutableMap["Swift"] = 9000
    // 用 put 方法放入 key-value 对
    mutableMap.put("OC" , 8600)
    // 下面输出{OC=8600, PHP=3400, Perl=4300, Ruby=5600, Go=5600, Swift=9000}
    println(mutableMap)
    // 删除 key 为"PHP"的 key-value 对
    mutableMap.remove("PHP")
    // 删除 key 为"Perl"的 key-value 对
    mutableMap.remove("Perl")
    // 下面输出：{OC=8600, Ruby=5600, Go=5600, Swift=9000}
    println(mutableMap)
    println(mutableMap.size)  // 输出 4
    // 删除所有元素
    mutableMap.clear()
    println(mutableMap)  // 输出{}
    println(mutableMap.size)  // 输出 0
}
```

上面程序中前两行粗体字代码分别调用了“[]”运算符、put()方法来放入 key-value 对。这两种语法的效果是一样的，但使用方括号的方式更简洁一些。

上面程序还调用了 clear()方法清空 Map 中所有的 key-value 对，调用该方法后即可看到该 Map 变成了空 Map，该 Map 的 size 属性也将返回 0。

5.6 本章小结

本章主要介绍了 Kotlin 的数组和三种集合类：Set、List 和 Map，其中数组用于按顺序存储相同类型的数据；Set 代表集合元素不允许重复的集合；List 代表元素有索引的集合；Map 则采用 key-value 对的形式存储数据，每项数据都由 key-value 对组成。因为数组、List 是按顺序来存储元素的，因此程序可以通过方括号“[]”运算符、索引来存取数组元素；而 Map 则是按 key-value 对的形式存储数据的，因此程序可通过方括号“[]”运算符、key 来存取 value。学习本章的重点是区分 Kotlin 集合与 Java 集合的差别：Kotlin 集合分为可变集合和不可变集合，Kotlin 的集合类提供了更多的方法。

CHAPTER

6

第6章 函数和Lambda表达式

本章要点

- 函数的语法和调用函数
- 递归函数
- 单表达式函数
- 命名参数
- 为形参指定默认值
- 尾递归函数
- 形参个数可变的函数
- 函数重载
- 局部函数的用法
- 函数类型
- 函数类型作为形参类型或返回值类型
- Lambda 表达式的基本用法
- Lambda 表达式的各种简化
- 匿名函数的功能和用法
- 捕获上下文中的变量和常量
- 内联函数的优势和缺陷
- 内联函数和禁止部分内联

函数是执行特定任务的一段代码。程序通过将一段代码定义成函数，并为该函数指定一个函数名，这样即可在需要的时候多次调用这段代码。因此，函数是代码复用的重要手段。

Kotlin 对 Java 的纯粹面向对象进行了弥补，增加了函数式编程的支持，这样提高了编程的灵活性。Kotlin融合了面向过程语言和面向对象语言的特征，因此Kotlin完全支持定义函数、调用函数。Kotlin 的函数比 C 语言的函数更加强大，Kotlin 支持局部函数（实际上局部函数是 Lambda 表达式的基础）。对于 Kotlin 开发者而言，函数是一个非常重要的知识点，尤其是对于普通的 Java 程序员来说，Kotlin 的函数可能是最需要花精力来掌握的内容。

与函数紧密相关的另一个知识点是 Lambda 表达式。Lambda 表达式可作为表达式、函数参数或函数返回值，因此使用 Lambda 表达式可以让程序更加简洁。

6.1 函数入门

函数就是 Kotlin 程序的重要组成单位，一个 Kotlin 程序可以由很多个函数组成。

6.1.1 定义和调用函数

在使用函数之前必须先定义函数。定义函数的语法格式如下：

```
fun 函数名(形参列表)[ : 返回值类型] {
    // 由零条到多条可执行语句组成的函数
}
```

Kotlin 声明函数必须使用 fun 关键字。函数语法格式的详细说明如下。

- **函数名**：从语法的角度来看，函数名只要是一个合法的标识符即可；从程序可读性的角度来看，函数名应该由一个或多个有意义的单词连缀而成，第一个单词首字母小写，后面每个单词首字母大写，其他字母全部小写，单词与单词之间不需要使用任何分隔符。
- **返回值类型**：返回值类型可以是 Kotlin 语言所允许的任何数据类型；如果声明了函数返回值类型，则函数体内应该有一条有效的 return 语句，该语句返回一个变量或一个表达式，这个变量或者表达式的类型必须与此处声明的类型匹配。如果希望声明一个函数没有返回值，则有如下两种声明方式。
  - 省略“: 返回值类型”部分。
  - 使用“: Unit”指定返回 Unit 代表没有返回值。



提示：

Kotlin 的 Unit 就相当于 Java 的 void。

- **形参列表**：形参列表用于定义该函数可以接受的参数，形参列表由零组到多组“形参名:参数类型”组合而成，多组参数之间以英文逗号（,）隔开，形参名和形参类型之间以英文冒号隔开。一旦在定义函数时指定了形参列表，那么在调用该函数时就必须传入对应的参数值——谁调用函数，谁负责为形参赋值。

函数体内多条可执行语句之间有严格的执行顺序，排在函数体前面的语句总是先执行，排在函数体后面的语句总是后执行。

下面程序定义了两个函数，并在程序中调用它们。

程序清单：codes\06\6.1\FunctionTest.kt

```
// 定义一个函数，声明两个形参，返回值为 Int 类型
fun max(x:Int , y:Int): Int {
    // 定义一个常量 z，该常量等于 x、y 中较大的值
    val z = if (x > y) x else y
    // 返回常量 z 的值
    return z
}
// 定义一个函数，声明一个形参，返回值为 String 类型
fun sayHi(name: String) : String {
    println("===正在执行 sayHi()函数===")
    return "${name}，您好！"
}
fun main(args: Array<String>){
    var a = 6
    var b = 9
    // 调用 max()函数，将函数返回值赋给 result 变量
    var result = max(a , b)  // ①
    println("result:${result}")
    // 调用 sayHi()函数，直接输出函数的返回值
    println(sayHi("孙悟空"))  // ②
}
```

上面程序中定义了两个函数：max()与 sayHi()，并在程序①、②号代码处分别调用了 max()和 sayHi()这两个函数。从下面的运行结果可以看出，当程序调用一个函数时，既可以把调用函数的返回值赋给指定变量，也可以将函数的返回值传给另一个函数，作为另一个函数的参数。

执行上面程序，将可以看到如下输出：

```
result:9
===正在执行 sayHi()函数===
孙悟空，您好！
```

从上面介绍不难看出，定义函数时必须为形参指定类型，调用函数时必须为形参传入参数值，而且传入的参数值必须与形参类型保持一致。如果传入的参数值与形参类型不匹配，编译时将会提示“类型不匹配”的错误。

如果为函数声明了返回值类型，则应该在函数体中使用 return 语句显式返回一个值，return 语句返回的值既可是常量，也可是有值的变量，还可是一个表达式。

例如上面的 max()函数，实际上也可简写为如下形式：

```
fun max(x:Int , y:Int) : Int {
    // 返回一个表达式
    return if (x > y) x else y
}
```

6.1.2 函数返回值和 Unit

如果希望明确指定函数没有返回值，则有如下两种方式。

- 直接省略“: 返回值类型”部分。
- 使用“: Unit”声明代表没有返回值。

如下程序采用了这两种方式来定义函数没有返回值。

程序清单：codes\06\6.1\NoReturn.kt

```
// 定义一个函数，该函数既无形参，也无返回值
fun foo() {
    println("程序执行 foo()函数")
}
// 定义一个函数，该函数没有返回值，但函数有一个形参
fun sayHi(name: String) : Unit {
    println("====程序执行 sayHi()函数====")
    println("${name}，你好。欢迎光临")
}
// 定义一个函数，该函数没有返回值，但函数有两个形参
fun showMsg(msg: String , count :Int) {
    for( i in 1 .. count) {
        println(msg)
    }
}
fun main(args: Array<String>) {
    foo()
    sayHi("孙悟空")
    showMsg("欢迎学习 Kotlin" , 3)
}
```

上面程序中粗体字代码分别示范了声明函数没有返回值的两种方式，正如前面程序中定义 main()函数没有返回值时都是直接省略了返回值类型声明——因为这种方式比较简单。所以，以后我们定义 Kotlin 函数时如果没有返回值，则只要直接省略返回值类型声明即可。

6.1.3 递归函数

在一个函数体内调用它自身，被称为函数递归。函数递归包含了一种隐式的循环，它会重复执行某段代码，但这种重复执行无须循环控制。

例如有如下数学题。已知有一个数列：$f(0) = 1$，$f(1)=4$，$f(n + 2) = 2*f(n+1) + f(n)$，其中 n 是大于 0 的整数，求 $f(10)$的值。这道题可以使用递归来求得。下面程序将定义一个 fn 函数，用于计算 $f(10)$的值。

程序清单：codes\06\6.1\Recursive.kt

```
fun fn(n:Int) : Int {
    if (n == 0) {
        return 1
    }
    else if (n == 1) {
        return 4
    }
    else {
        // 在函数中调用它自身，就是函数递归
        return 2 * fn(n - 1) + fn(n - 2)
    }
}
fun main(args: Array<String>) {
    // 输出 fn(10)的结果
    println("fn(10)的结果是:${fn(10)}")
}
```

在上面的 fn 函数体中，再次调用了 fn 函数，这就是函数递归。注意在 fn 函数中调用 fn

的形式：

```
return 2 * fn(n - 1) + fn(n - 2)
```

对于 fn(10)，即等于 2 * fn(9) + fn(8)，其中 fn(9)又等于 2 * fn(8) + fn(7)……依此类推，最终会计算到 fn(2)等于 2 * fn(1) + fn(0)，即 fn(2)是可计算的，这样递归带来的隐式循环就有结束的时候，然后一路反算回去，最后就可以得到 fn(10)的值。

仔细看上面递归的过程，当一个函数不断地调用它自身时，必须在某个时刻函数的返回值是确定的，即不再调用它自身；否则，这种递归就变成了无穷递归，类似于死循环。因此，在定义递归函数时有一条最重要的规定：递归一定要向已知方向进行。

例如，如果把上面数学题改为如此。已知有一个数列：$f(20) = 1$，$f(21)=4$，$f(n + 2) = 2 * f(n+1) + f(n)$，其中 n 是大于 0 的整数，求 $f(10)$的值。那么 fn 的函数体就应该改为如下形式：

```
fun fn(n: Int) : Int {
    if (n == 20) {
        return 1
    }
    else if (n == 21) {
        return 4
    }
    else {
        // 在函数中调用它自身，就是函数递归
        return fn(n + 2) - 2 * fn(n + 1)
    }
}
```

从上面的 fn 函数来看，当程序要计算 fn(10)的值时，fn(10)等于 fn(12)- 2 * fn(11)，而 fn(11)等于 fn(13)- 2 * fn(12)……依此类推，直到 fn(19)等于 fn(21)- 2 * fn(20)，此时就可以得到 fn(19)的值，然后依次反算到 fn(10)的值。这就是递归的重要规则：对于求 fn(10)而言，如果 fn(0)和 fn(1)是已知的，则应该采用 fn(n) = 2 * fn(n - 1) + fn(n - 2)的形式递归，因为小的一端已知；如果 fn(20)和 fn(21)是已知的，则应该采用 fn(n) = fn(n + 2) - 2 * fn(n + 1)的形式递归，因为大的一端已知。

递归是非常有用的，例如程序希望遍历某个路径下的所有文件，但这个路径下的文件夹的深度是未知的，那么就可以使用递归来实现这个需求。系统可定义一个函数，该函数接受一个文件路径作为参数，该函数可遍历出当前路径下的所有文件和文件路径——在该函数中再次调用函数本身来处理该路径下的所有文件路径。

总之，只要在一个函数的函数体实现中再次调用了函数本身，就是递归函数。递归一定要向已知方向进行。

6.1.4 单表达式函数

在某些情况下，函数只是返回单个表达式，此时可以省略花括号并在等号（=）后指定函数体即可。这种方式被称为单表达式函数。

例如，下面定义了一个计算矩形面积的函数。

程序清单：codes\06\6.1\SingleExprTest.kt

```
fun area(x: Double, y: Double): Double = x * y
fun main(args: Array<String>) {
```

```
    // 调用 area 函数
    println(area(2.0 , 5.0))
}
```

对于单表达式函数而言，编译器可以推断出函数的返回值类型，因此 Kotlin 允许省略声明函数的返回值类型。例如，上面的粗体字函数可改为如下形式：

```
fun area(x: Double, y: Double) = x * y
```

6.2 函数的形参

Kotlin 函数的功能比 Java 方法更加强大，其中一点就体现在函数的形参上，Kotlin 函数的形参功能更加丰富，而且更加灵活。下面详细介绍 Kotlin 函数的形参。

6.2.1 命名参数

Kotlin 函数的参数名不是无意义的，Kotlin 允许调用函数时通过名字来传入参数值。因此，Kotlin 函数的参数名应该具有更好的语义——程序可以立刻明确传入函数的每个参数的含义。

Kotlin 函数除第一个参数之外，其他所有形参都分配隐式的外部形参名——这些外部形参名与内部形参名保持一致。例如如下程序。

程序清单：codes\06\6.2\NamedParamTest.swift

```
// 定义一个函数
fun girth(width: Double , height: Double): Double{
    println("width: ${width}")
    println("height: ${height}")
    return 2 * (width + height)
}
fun main(args: Array<String>) {
    // 传统调用函数的方式，根据位置传入参数
    println(girth(3.5, 4.8))
    // 根据参数名来传入参数
    println(girth(width = 3.5, height = 4.8))
    // 使用命名参数时可交换位置
    println(girth(height = 4.8, width = 3.5))
    // 部分使用命名参数，部分使用位置参数
    println(girth(3.5, height = 4.8))
}
```

上面程序定义了一个简单的 girth()函数，该函数包含 width 和 height 两个参数，这个函数与前面定义的函数并没有任何区别。

接下来调用该函数时，既可使用传统的根据位置参数来调用（如上面程序中第二行粗体字代码所示），也可根据命名参数来调用（如上面程序中第三行粗体字代码所示），并可交换参数的位置（如上面程序中第四行粗体字代码所示），还可混合使用命名参数和位置参数（如上面程序中第五行粗体字代码所示）。

需要说明的是，如果希望调用函数时混合使用命名参数和位置参数，那么命名参数必须位于位置参数之后。换句话说，在命名参数之后的只能是命名参数。例如，如下代码是错误的。

```
// 位置参数必须放在命名参数之前，下面代码错误
println(girth(width = 3.5, 4.8))
```

6.2.2 形参默认值

在某些情况下，程序需要在定义函数时为一个或多个形参指定默认值——这样调用函数时就可以省略该形参，而直接使用该形参的默认值。

为形参指定默认值的语法格式如下：

```
形参名: 形参类型 = 默认值
```

从上面的语法格式可以看出，形参的默认值紧跟在形参类型之后，中间以英文等号“=”隔开。

例如，如下程序为 name、message 形参指定了默认值。

程序清单：codes\06\6.2\DefaultParamTest.kt

```
// 为两个参数指定默认值
fun sayHi(name: String = "孙悟空",
    message: String = "欢迎来到疯狂软件"){
    println("${name} , 您好")
    println("消息是: ${message}")
}
fun main(args: Array<String>) {
    // 全部使用默认参数
    sayHi()
    // 只有 message 参数使用默认值
    sayHi("白骨精")
    // 两个参数都不使用默认值
    sayHi("白骨精", "欢迎学习 Kotlin")
    // 只有 name 参数使用默认值
    sayHi(message = "欢迎学习 Kotlin")
}
```

上面程序中定义 sayHi()函数时为 name、message 形参指定了默认值，因此程序中第一行粗体字代码调用 sayHi()函数时没有为 name、message 参数指定参数值，此时 name、message 参数将会使用它的默认值；程序第二次调用 sayHi()函数时为 name 参数（使用位置参数）指定了参数值，此时 message 参数将会使用默认值；程序第三次调用 sayHi()函数时为 name、message 参数（使用位置参数）都指定了参数值，因此这两个参数都使用开发者传入的参数值；程序第四次调用 sayHi()函数时只为 message 参数（使用命名参数）传入参数值，此时 name 参数将使用默认值。

运行上面程序，可看到如下输出：

```
孙悟空 , 您好
消息是: 欢迎来到疯狂软件
白骨精 , 您好
消息是: 欢迎来到疯狂软件
白骨精 , 您好
消息是: 欢迎学习 Kotlin
孙悟空 , 您好
消息是: 欢迎学习 Kotlin
```

从上面程序中可以看出，如果只传入一个位置参数，由于该参数位于第一位，系统会将该参数值传给 name 参数。因此，开发者按如下形式调用 sayHi()函数时，传入的"欢迎学习 Kotlin"字符串将传给 name 参数，而不是 message 参数。

```
sayHi("欢迎学习 Kotlin")
```

我们也不能按如下形式来调用 sayHi()函数。因为 Kotlin 规定：命名参数必须位于位置参数的后面。

```
sayHi(name="白骨精", "欢迎学习 Kotlin")
```

那么，能不能单纯地将上面两个参数交换位置呢？

```
sayHi("欢迎学习 Kotlin" , name="白骨精")
```

上面调用依然是错误的，因为第一个字符串没有指定命名参数，将使用位置参数为 name 参数传入参数值，第二个参数使用了命名参数为 name 参数传入参数值，这意味着两个参数值其实都会传给 name 参数，因此是错误的。

函数调用改为如下两种形式是正确的：

```
sayHi("白骨精", message="欢迎学习 Kotlin")
sayHi(name="白骨精", message="欢迎学习 Kotlin")
```

上面第一行代码先使用位置参数为 name 参数传入参数值，再使用命名参数为 message 参数传入参数值；上面第二行代码中的 name、message 参数都使用命名参数传入参数值。

通过为函数形参指定默认值，可以减少函数重载的数量。例如，Java 的 String 类包含如下两个方法。

- substring(start, end)：截取字符串从 start 到 end 之间的子串。
- substring(start)：截取字符串从 start 到结束处的子串。

使用参数默认值之后，上面两个方法可合并为一个方法：substring(start, end=this.length())。

根据上面介绍不难发现，如果在定义函数之后将有默认值的参数放在普通参数的前面，并且不想为默认参数重新传入参数值，那么就只能使用命名参数为其他参数传入参数值。例如如下代码。

程序清单：codes\06\6.2\DefaultParamTest2.kt

```
// 定义一个打印三角形的函数，只为第一个参数指定默认值
fun printTriangle(height: Int = 5, char: Char){
    for (i in 1..height) {
        // 先打印一排空格
        for (j in 0 until height - i) {
            print(" ");
        }
        // 再打印一排特殊字符
        for (j in 0 until 2 * i - 1) {
            print(char);
        }
        println()
    }
}
fun main(args: Array<String>) {
    printTriangle(6, '@')
    printTriangle(7, char = '#')
    printTriangle(char = '*')
}
```

上面程序定义了一个 printTriangle()函数，该函数的第一个 height 参数有默认值，第二个参数 char 没有默认值。

在上面的 main()函数中，程序第一次调用 printTriangle()时，使用了两个位置参数分别为 height、char 传入参数值，这当然是允许的；程序第二次调用 printTriangle()时，第一个参数使用了位置参数，那么该参数值将传给 height 参数，第二个参数使用了命名参数为 char 参数传入参数值，这也是允许的；第三次调用 printTriangle()时，只使用命名参数为 char 参数传入参数值，此时 height 参数将使用默认值，这是符合语法的。

但如果想让 height 参数使用默认值，且使用位置参数为 char 参数传入参数值，那么就会变成如下形式：

```
printTriangle('#')
```

上面调用形式是错误的，因为在调用 printTriange 时只传入一个参数，该参数（第一个参数）值将会传给 height 参数，而不是传给 char 参数。

由此可见，如果将带默认值的参数定义在无默认值的参数之前，那么想为默认参数使用默认值时，只能使用命名参数为无默认值的参数传入参数值。出于这种考虑，Kotlin 建议将带默认值的参数放在函数形参列表的后面。上面示例将带默认值的 height 形参作为形参列表的第一个参数，这并不是一种好的编程习惯。

注意：

通常建议将带默认值的参数定义在形参列表的最后。



有一种情况例外，如果最后一个 Lambda 表达式参数从括号外传给函数调用，那么才允许默认参数不传值。关于 Lambda 表达式作为参数值的情形，本章后面会有更详细的介绍。

6.2.3 尾递归函数

Kotlin 还支持一种尾递归函数的编程方式，当函数将调用自身作为它执行的最后一行代码，且递归调用后没有更多代码时，可使用尾递归语法。

另外，尾递归不能在异常处理的 try、catch、finally 块中使用。

尾递归函数需要使用 tailrec 修饰。

例如，有如下计算阶乘的函数。

程序清单：codes\06\6.2\Trailrec.kt

```
// 定义计算阶乘的函数
fun fact(n: Int): Int{
    if (n == 1){
        return 1
    } else {
        return n * fact(n - 1)
    }
}
```

上面函数将调用自身作为其执行体的最后一行代码，且递归调用后没有更多代码，因此可以将该函数改为尾递归语法。此时，上面函数可改为如下形式（程序清单同上）：

```
// 使用尾递归函数的语法
tailrec fun factRec(n: Int, total: Int = 1): Int =
    if (n == 1) total else factRec(n - 1, total * n)
```

与普通递归相比，编译器会对尾递归进行修改，将其优化成一个快速而高效的基于循环的版本，这样就可以减少可能对内存的消耗。

6.2.4 个数可变的形参

Kotlin 允许定义个数可变的参数，从而允许为函数指定数量不确定的形参。如果在定义函数时，在形参的类型前添加 vararg 修饰，则表明该形参可以接受多个参数值，多个参数值被当成数组传入。下面程序定义了一个形参个数可变的函数。

程序清单：codes\06\6.2\Varargs.kt

```
// 定义了形参个数可变的函数
fun test(a : Int , vararg books : String) {
    // books 被当成数组处理
    for( b in books) {
        println(b)
    }
    // 输出整型变量 a 的值
    println(a)
}
fun main(args: Array<String>) {
    // 调用 test()函数
    test(5 , "疯狂 iOS 讲义" , "疯狂 Android 讲义")
}
```

运行上面程序，将看到如下运行结果：

```
疯狂 iOS 讲义
疯狂 Android 讲义
5
```

从上面的运行结果可以看出，当调用 test()函数时，books 参数可以传入多个字符串作为参数值。从 test()的函数体代码来看，形参个数可变的参数本质就是一个数组参数。

Kotlin 允许个数可变的形参可以处于形参列表的任意位置（不要求是形参列表的最后一个参数），但 Kotlin 要求一个函数最多只能带一个个数可变的形参。例如如下程序。

程序清单：codes\06\6.2\Varargs2.swift

```
// 定义了形参个数可变的函数
fun test(vararg books : String , num: Int) {
    // books 被当成数组处理
    for (i in books.indices) {
        println(books[i])
    }
    println(num)
}
fun main(args: Array<String>) {
    // 调用 test()函数
    test("疯狂 iOS 讲义", "疯狂 Android 讲义", num = 20)
}
```

正如上面程序中所看到的，test()函数的第一个参数就是个数可变的形参。由于该参数可接受个数不等的参数值，因此，如果需要给后面的参数传入参数值，则必须使用命名参数。

如果我们已有一个数组，程序希望将数组的多个元素传给个数可变的参数，则可以在传入的数组参数前添加“*”运算符。例如如下代码：

```
var arr = arrayOf("疯狂 Kotlin 讲义", "疯狂 Java 讲义")
// 将数组的多个元素传给个数可变的参数
test(*arr, num = 20)
```

6.3 函数重载

与 Java 类似，Kotlin 允许定义多个同名函数，只要形参列表或返回值类型不同就行。如果程序包含了两个或两个以上函数名相同，但形参列表不同的函数，就被称为函数重载。

与 Java 类似的是，Kotlin 的函数重载也只能通过形参列表进行区分，形参个数不同、形参类型不同都可以算函数重载。但仅有形参名不同、返回值类型不同或修饰符不同，则不能算函数重载。

下面程序中包含了函数重载的示例。

程序清单：codes\06\6.3\OverloadTest.kt

```
// 下面定义了 3 个 test()函数，但函数的形参列表不同
// 系统可以区分它们，这被称为函数重载
fun test() {
    println("无参数的 test()函数")
}
// 该函数类型为(String): Unit
fun test(msg:String) {
    println("重载的 test()函数${msg}")
}
// 该函数类型为(Int): String
fun test(msg: Int): String {
    println("重载的 test()函数${msg},带返回值")
    return "test"
}
fun main(args: Array<String>){
    // 调用 test()时没有传入参数，因此系统调用上面没有参数的 test()函数
    test()
    // 调用带 String 参数的 test()函数
    test("fkjava")
    // 调用带 Int 参数的 test()函数，该函数返回字符串
    var rt = test(30)
    println(rt)
}
```

编译、运行上面程序完全正常。上面 3 个 test()函数的函数名虽然相同，但因为它们的形参列表不同，所以系统可以正常区分它们。

如果被重载的函数中包含了个数可变的形参，则尤其需要注意。看下面程序中定义的两个重载的函数。

程序清单：codes\06\6.3\OverloadVarargs.kt

```
fun test(msg: String) {
    println("只有一个字符串参数的 test()函数 ${msg}")
}
// 因为前面已经有了带一个字符串参数的 test()函数
// 此处的个数可变的形参又可以包含带一个字符串参数的形式
fun test(vararg books: String) {
    println("****形参个数可变的 test()函数**** ${books.contentToString()}")
```

```
}
fun main(args: Array<String>) {
    // 下面两次调用将执行第二个 test()函数
    test()
    test("aa" , "bb")
    test("aa")  // ①
}
```

编译、运行上面程序，将看到 test()和 test("aa" , "bb")两次调用的都是 test(books:String...)函数。

对于①号粗体字代码：test("aa")，这行代码才会调用 test(msg:String)函数，这意味着虽然这行代码既可匹配 test(String)函数，也可匹配 test(varargs String)函数，但 Kotlin 会尽量执行最精确的匹配，因此 Kotlin 会这样调用匹配 test(String)函数。



提示：

大部分时候并不推荐重载形参个数可变的函数，因为这样做确实没有太大的意义，而且容易导致错误。

6.4 局部函数

前面所看到的函数都是在全局范围内定义的，它们都是全局函数。Kotlin 还支持在函数体内部定义函数，这种被放在函数体内定义的函数称为局部函数。

在默认情况下，局部函数对外部是隐藏的，局部函数只能在其封闭（enclosing）函数内有效，其封闭函数也可以返回局部函数，以便程序在其他作用域中使用局部函数。

程序清单：codes\06\6.4\LocalFunctionTest.kt

```
// 定义函数，该函数的返回值类型为 Int
fun getMathFunc(type: String , nn: Int) : Int {
    // 定义一个计算平方的局部函数
    fun square(n: Int) : Int {  // ①
        return n * n
    }
    // 定义一个计算立方的局部函数
    fun cube(n: Int) : Int {  // ②
        return n * n * n
    }
    // 定义一个计算阶乘的局部函数
    fun factorial(n: Int) : Int {  // ③
        var result = 1
        for (index in 2 .. n) {
            result *= index
        }
        return result
    }
    when(type) {
        // 调用局部函数
        "square" -> return square(nn)
        "cube" -> return cube(nn)
        else -> return factorial(nn)
    }
```

```
}
fun main(args: Array<String>) {
    println(getMathFunc("square", 3)) // 输出 9
    println(getMathFunc("cube", 3)) // 输出 27
    println(getMathFunc("", 3)) // 输出 6
}
```

上面程序中第一行粗体字代码定义了 getMathFunc()函数的返回值类型为 Int，接下来程序的①②③号粗体字代码定义了 3 个局部函数，而 getMathFunc()函数则根据参数选择调用不同的局部函数。

如果封闭函数没有局部函数返回，那么局部函数将只能在封闭函数内部调用，如上面程序所示。

另外，还会出现一种情况，如果封闭函数将局部函数返回，且程序使用变量保存了封闭函数的返回值，那么这些局部函数的作用域就会被扩大，因此程序完全可以自由地调用它们，就像它们都是全局函数一样。

下一节我们先介绍函数类型，然后就会介绍函数返回函数的情况。

6.5 高阶函数

Kotlin 不是纯粹的面向对象语言，Kotlin 的函数也是一等公民，因此函数本身也具有自己的类型。函数类型就像前面介绍的数据类型一样，既可用于定义变量，也可用作函数的形参类型，还可作为函数的返回值类型。

6.5.1 使用函数类型

Kotlin 的每个函数都有特定的类型，函数类型由函数的形参列表、->和返回值类型组成。例如如下函数：

```
func foo(a : Int , name:String) -> String{
    ...
}
```

该函数的形参列表、->和返回值类型为(Int , String) -> String，这就是该函数的类型。

```
func bar(width:Double , height: Double){
    ...
}
```

该函数的形参列表、->和返回值类型为(Double , Double) -> Unit 或(Double , Double)，这就是该函数的类型。

```
func test(){
    ...
}
```

该函数的形参列表、->和返回值类型为() ->Unit 或()，这就是该函数的类型。

掌握了函数类型之后，接下来就可以使用函数类型定义变量了——就像使用普通类型定义变量一样。例如如下代码。

程序清单：codes\06\6.5\FunctionTypeTest.kt

```
// 定义一个变量，其类型为(Int , Int) -> Int
```

```
var myfun : (Int , Int) -> Int
// 定义一个变量，其类型为(String)
var test : (String)
```

定义了函数类型的变量之后，接下来程序即可将函数赋值给该变量。例如如下程序（程序清单同上）：

```
// 定义一个计算乘方的函数
fun pow(base: Int , exponent:Int) : Int {
    var result = 1
    for( i in 1 .. exponent) {
        result *= base
    }
    return result
}
// 将 pow 函数赋值给 myfun，则 myfun 可当成 pow 使用
myfun = ::pow
println(myfun(3 , 4)) // 输出 81
// 定义一个计算面积的函数
fun area(width: Int, height:Int) : Int {
    return width * height
}
// 将 area 函数赋值给 myfun，则 myfun 可当成 area 使用
myfun = ::area
println(myfun(3, 4)) // 输出 12
```

从上面代码可以看出，程序依次将 pow()、area()函数赋值给 myfun 变量——只要被赋值的函数类型与 myfun 的变量类型一致，程序就可以赋值成功。

当直接访问一个函数的函数引用，而不是调用函数时，需要在函数名前添加两个冒号，而且不能在函数后面添加圆括号——一旦添加圆括号，就变成了调用函数，而不是访问函数引用。

通过使用函数类型的变量，可以让 myfun 在不同的时间指向不同的函数，从而让程序更加灵活。由此可见，使用函数类型的好处是让程序更加灵活。

除此之外，程序还可使用函数类型作为形参类型和返回值类型。

6.5.2 使用函数类型作为形参类型

有时候需要定义一个函数，该函数的大部分计算逻辑都能确定，但某些处理逻辑暂时无法确定——这意味着某些程序代码需要动态改变，如果希望调用函数时能动态传入这些代码，就需要在函数中定义函数类型的形参，这样即可在调用该函数时传入不同的函数作为参数，从而动态改变这些代码。

Kotlin 支持像使用其他类型一样使用函数类型，因此完全可以在函数中定义函数类型的形参。例如如下程序。

程序清单：codes\06\6.5\FunctionParamTest.kt

```
// 定义函数类型的形参，其中 fn 是(Int) -> Int 类型的形参
fun map(data : Array<Int>, fn: (Int) -> Int) : Array<Int> {
    var result = Array<Int>(data.size, {0})
    // 遍历 data 数组的每个元素，并用 fn 函数对 data[i]进行计算
    // 然后将计算结果作为新数组的元素
    for (i in data.indices) {
        result[i] = fn(data[i])
    }
```

```
    return result
}
// 定义一个计算平方的函数
fun square(n: Int) : Int {
    return n * n
}
// 定义一个计算立方的函数
fun cube(n: Int) : Int {
    return n * n * n
}
// 定义一个计算阶乘的函数
fun factorial(n: Int) : Int {
    var result = 1
    for (index in 2 .. n) {
        result *= index
    }
    return result
}
fun main(args: Array<String>) {
    var data = arrayOf(3 , 4 , 9 , 5, 8)
    println("原数据${data.contentToString()}")
    // 下面程序代码调用 map()函数 3 次，每次调用时都传入不同的函数
    println("计算数组元素的平方")
    println(map(data , ::square).contentToString())
    println("计算数组元素的立方")
    println(map(data , ::cube).contentToString())
    println("计算数组元素的阶乘")
    println(map(data , ::factorial).contentToString())
}
```

上面程序中定义了一个 map()函数，该函数的第二个参数是一个函数类型的形参，这意味着每次调用函数时都可动态传入一个函数，随着实际传入函数的改变，就可以动态改变 map()函数中的部分计算代码。

接下来主程序的 3 行粗体字代码调用了 map()函数 3 次，3 次调用时依次传入了 square、cube、factorial 函数作为参数，这样每次调用 map()函数时实际的执行代码都是有区别的。

编译、运行上面程序，可以看到如下输出：

```
原数据[3, 4, 9, 5, 8]
计算数组元素的平方
[9, 16, 81, 25, 64]
计算数组元素的立方
[27, 64, 729, 125, 512]
计算数组元素的阶乘
[6, 24, 362880, 120, 40320]
```

从上面介绍不难看出，定义了函数类型的形参后，就可以在调用函数时动态地传入函数——实际上可以动态地改变被调用函数的部分代码。

这种用法其实就是“命令模式”，如果读者熟悉 Java 编程的知识，则不难发现这种用法与 Java 的 Lambda 表达式的用法大同小异。

6.5.3 使用函数类型作为返回值类型

前面已经提到，Kotlin 还支持定义函数类型的返回值，这样即可将其他函数作为函数的返

回值。例如如下程序。

程序清单：codes\06\6.5\FunctionReturnTest.kt

```
// 定义函数，该函数的返回值类型为(Int) -> Int
fun getMathFunc(type: String): (Int) -> Int {
    // 定义一个计算平方的局部函数
    fun square(n: Int) : Int {  // ①
        return n * n
    }
    // 定义一个计算立方的局部函数
    fun cube(n: Int) : Int {  // ②
        return n * n * n
    }
    // 定义一个计算阶乘的局部函数
    fun factorial(n: Int) : Int {  // ③
        var result = 1
        for (index in 2 .. n) {
            result *= index
        }
        return result
    }
    when(type) {
        // 返回局部函数
        "square" -> return ::square
        "cube" -> return ::cube
        else -> return ::factorial
    }
}
fun main(args: Array<String>) {
    // 调用 getMathFunc()，程序返回一个(Int)->Int 类型的函数
    var mathFunc = getMathFunc("cube") // 得到 cube 函数
    println(mathFunc(5)) // 输出 125
    mathFunc = getMathFunc("square") // 得到 square 函数
    println(mathFunc(5)) // 输出 25
    mathFunc = getMathFunc("other") // 得到 factorial 函数
    println(mathFunc(5)) // 输出 120
}
```

上面程序中的第一行粗体字代码定义了一个 getMathFunc()函数，该函数的返回值类型为(Int) -> Int，这是一个函数类型，也就是 getMathFunc()函数将会返回一个类型为(Int) -> Int 的函数。接下来 getMathFunc()函数体内的①②③号粗体字代码分别定义了 3 个局部函数，这 3 个局部函数都是(Int)->Int 类型的，然后 getMathFunc()函数就使用这 3 个局部函数作为返回值。

一旦定义了返回值类型为(Int) -> Int 的 getMathFunc()函数，接下来程序调用 getMathFunc()函数时就可以返回(Int) -> Int 类型的函数。上面程序中的最后 3 行粗体字代码通过调用 getMathFunc()函数，分别得到了 3 个不同的(Int) -> Int 函数。

6.6 局部函数与 Lambda 表达式

Lambda 表达式是现代编程语言争相引入的一种语法，如果说函数是命名的、方便复用的代码块，那么 Lambda 表达式则是功能更灵活的代码块，它可以在程序中被传递和调用。

6.6.1 回顾局部函数

回顾 6.4 节介绍的 LocalFunctionTest.kt 程序，该程序中的 getMathFunc()函数将会返回 3 个局部函数中的一个。该函数代码如下：

```
// 定义函数，该函数的返回值类型为(Int) -> Int
fun getMathFunc(type type:String) -> (Int) -> Int {
    // 定义 3 个局部函数
    ...
    // 返回局部函数
    when(type) {
        // 返回局部函数
        "square" -> return ::square
        "cube" -> return ::cube
        else -> return ::factorial
    }
}
```

由于局部函数的作用域默认仅停留在其封闭函数之内，因此这 3 个局部函数的函数名的作用太有限了——仅仅就是在程序的 when 表达式中作为返回值使用。一旦离开了 getMathFunc()函数体，这 3 个局部函数的函数名就失去了意义。

既然局部函数的函数名没有太大的意义，那么就考虑使用 Lambda 表达式来简化局部函数的写法。

6.6.2 使用 Lambda 表达式代替局部函数

如果使用 Lambda 表达式来简化 LocalFunctionTest.kt，则可以将程序改写成如下形式。

程序清单：codes\06\6.6\LambdaTest.kt

```
// 定义函数，该函数的返回值类型为(Int) -> Int
fun getMathFunc(type: String) : (Int) -> Int {
    // 该函数返回的是 Lambda 表达式
    when(type) {
        "square" -> return {n: Int -> // ①
                n * n
            }
        "cube" -> return {n: Int -> // ②
                n * n * n
            }
        else -> return {n: Int -> // ③
                var result = 1
                for ( index in 2..n ){
                    result *= index
                }
                result
            }
    }
}
fun main(args: Array<String>) {
    // 调用 getMathFunc()，程序返回一个(Int)->Int 类型的函数
    var mathFunc = getMathFunc("cube") // 得到 cube 函数
    println(mathFunc(5)) // 输出 125
    mathFunc = getMathFunc("square") // 得到 square 函数
    println(mathFunc(5)) // 输出 25
```

```
    mathFunc = getMathFunc("other") // 得到 factorial 函数
    println(mathFunc(5)) // 输出 120
}
```

将上面粗体字代码中 return 后面的部分与 LocalFunctionTest.lt 中的 square()、cube()、factorial()局部函数的代码进行对比，不难发现定义 Lambda 表达式与局部函数的代码几乎是一样的。定义 Lambda 表达式与局部函数只是存在如下区别。

- Lambda 表达式总是被大括号括着。
- 定义 Lambda 表达式不需要 fun 关键字，无须指定函数名。
- 形参列表（如果有的话）在 -> 之前声明，参数类型可以省略。
- 函数体（Lambda 表达式执行体）放在->之后。
- 函数的最后一个表达式自动被作为 Lambda 表达式的返回值，无须使用 return 关键字。

6.6.3 Lambda 表达式的脱离

作为函数参数传入的 Lambda 表达式可以脱离函数独立使用。例如如下程序。

程序清单：codes\06\6.6\EscapeTest.kt

```
// 定义一个 List 类型的变量，并将其初始化为空 List
var lambdaList = java.util.ArrayList<(Int) -> Int>()
// 定义一个函数，该函数的形参类型为函数
fun collectFn(fn : (Int) -> Int ) {
    // 将传入的 fn 参数（函数或 Lambda 表达式）添加到 lambdaList 集合中
    // 这意味着 fn 将可以在 collectFn 范围之外使用
    lambdaList.add(fn)
}
fun main(args: Array<String>) {
    // 调用 collectFn()函数两次，将会向 lambdaList 中添加元素（每个元素都是 Lambda 表达式）
    collectFn({it * it})
    collectFn({it * it * it})
    // 输出 lambdaList 的长度
    println(lambdaList.size)
    // 依次调用 lambdaList 集合的元素（每个元素都是 Lambda 表达式）
    for( i in lambdaList.indices ) {
        println( lambdaList[i](i + 10) ) // ①
    }
}
```

上面程序中定义了一个 collectFn()函数，该函数带一个函数类型的形参，在 collectFn()函数内部只是将传入的 fn 参数添加到 lambdaList 集合中，这意味着程序接下来可通过 lambdaList 集合访问传给 collectFn()函数的 Lambda 表达式——程序中①号粗体字代码正是通过 lambdaList 集合元素来访问 Lambda 表达式的。

从上面程序可以看出，把 Lambda 表达式作为参数传给 collectFn()函数之后，这些 Lambda 表达式可以脱离 collectFn()函数使用。

6.7 Lambda 表达式

Kotlin 提供了简洁的 Lambda 表达式语法，正如从上面程序中所看到的。Lambda 表达式的标准语法如下：

```
{(形参列表) ->
    // 零条到多条可执行语句
}
```

正如前面所介绍的，Lambda 表达式的语法与局部函数非常相似，因此 Lambda 表达式的形参列表的语法以及执行体与局部函数几乎相同。Lambda 表达式的语法与局部函数的主要区别就是前面提到的几点。

此外，Lambda 表达式还可做进一步的优化，从而使得程序更加简洁。

6.7.1 调用 Lambda 表达式

Lambda 表达式的本质是功能更灵活的代码块，因此完全可以将 Lambda 表达式赋值给变量或直接调用 Lambda 表达式。例如如下程序。

程序清单：codes\06\6.7\InvokeLambda.kt

```
fun main(args: Array<String>) {
    // 定义一个 Lambda 表达式，并将它赋值给 square 变量
    var square = {n: Int ->
        n * n
    }
    // 使用 square 调用 Lambda 表达式
    println(square(5))  // 输出 25
    println(square(6))  // 输出 36
    // 定义一个 Lambda 表达式，并在它后面添加圆括号来调用该 Lambda 表达式
    var result = {base: Int , exponent:Int ->
        var result = 1
        for (i in 1 .. exponent) {
            result *= base
        }
        result
    }(4, 3)
    println(result)  // 输出 64
}
```

上面程序中定义了两个Lambda表达式，其中第一个Lambda表达式被赋值给square变量，因此程序以后就可以通过该变量重复调用该 Lambda 表达式，如程序中前两行粗体字代码所示。

程序的第二个 Lambda 表达式并未赋值给任何变量，也没有将它传给任何函数或方法，因此程序只能在定义该 Lambda 表达式的同时调用它。正如从上面程序中所看到的，程序在第二个 Lambda 表达式的后面使用圆括号执行调用，并传入了对应的参数，这样 result 变量得到的不是 Lambda 表达式，而是执行它的返回值。

6.7.2 利用上下文推断类型

完整的 Lambda 表达式需要定义形参类型，但是如果 Kotlin 可以根据 Lambda 表达式上下文推断出形参类型，那么 Lambda 表达式就可以省略形参类型。

因此，可以使用如下程序来简化上面的程序。

程序清单：codes\06\6.7\OmitType.kt

```
fun main(args: Array<String>) {
    // 由于程序定义了 square 变量的类型
    // 因此 Kotlin 可以推断出 Lambda 表达式的形参类型
```

```
    // 所以 Lambda 表达式可以省略形参类型
    var square:(Int) -> Int = {n -> n * n}
    // 使用 square 调用 Lambda 表达式
    println(square(5))  // 输出 25
    println(square(6))  // 输出 36
    // 此时 Kotlin 无法推断出 base、exponent 两个形参的类型
    // 因此必须为其指定类型
    var result = {base: Int, exponent: Int ->
        var result = 1
        for ( i in 1 .. exponent) {
            result *= base
        }
        result
    }(4, 3)
    println(result)  // 输出 64
    var list = listOf("Java", "Kotlin", "Go")
    // 使用 Lambda 表达式定义去除条件：长度大于 3 的集合元素被去除
    // 由于 doWhile()方法的形参是(T) -> Boolean 类型
    // 因此调用该方法时可省略形参类型
    var rt = list.dropWhile({e -> e.length > 3})
    println(rt) // 输出[Go]
}
```

上面程序中第一个 Lambda 表达式被赋值给(Int) -> Int 类型的变量，这样 Kotlin 可以很容易地推断出该 Lambda 表达式的形参类型，因此 Lambda 表达式可以省略形参类型。第二个 Lambda 表达式后面紧跟着(4,3)，Kotlin 无法准确推断出该 Lambda 表达式的形参类型，因此需要显式声明 Lambda 表达式的形参类型；第三行粗体字代码调用了 list 的 dropWhile()方法，该方法的形参是(T) -> Boolean 类型，因此 Kotlin 可以推断出该 Lambda 表达式的形参类型就是集合元素的类型。

6.7.3 省略形参名

Lambda 表达式不仅可以省略形参类型，而且如果只有一个形参，那么 Kotlin 允许省略 Lambda 表达式的形参名。如果 Lambda 表达式省略了形参名，那么此时->也不需要了，Lambda 表达式可通过 it 来代表形参。

例如，如下程序更加简洁。

程序清单：codes\06\6.7\OmitParam.kt

```
fun main(args: Array<String>) {
    // 省略形参名，用 it 代表形参
    var square:(Int) -> Int = {it * it}
    // 使用 square 调用 Lambda 表达式
    println(square(5))  // 输出 25
    println(square(6))  // 输出 36
    // Lambda 表达式有两个形参，无法省略
    var result = {base: Int, exponent: Int ->
        var result = 1
        for ( i in 1 .. exponent) {
            result *= base
        }
        result
    }(4, 3)
    println(result)  // 输出 64
```

```
    var list = listOf("Java", "Kotlin", "Go")
    // 省略形参名，用 it 代表形参
    var rt = list.dropWhile({it.length > 3})
    println(rt) // 输出[Go]
}
```

6.7.4 调用 Lambda 表达式的约定

Kotlin 语言有一个约定：如果函数的最后一个参数是函数类型，而且你打算传入一个 Lambda 表达式作为相应的参数，那么就允许在圆括号之外指定 Lambda 表达式。



提示：

其实这种用法也不是 Kotlin 独有的，在其他语言中这种用法被称为“尾随闭包（Tail Closure）”。

例如如下代码。

程序清单：codes\06\6.7\TailLambda.kt

```
fun main(args: Array<String>) {
    var list = listOf("Java", "Kotlin", "Go")
    // 最后一个参数是 Lambda 表达式，可将表达式写在圆括号外面
    var rt = list.dropWhile(){it.length > 3}
    println(rt) // 输出[Go]
    var map = mutableMapOf("疯狂 Android 讲义" to 56)
    // 最后一个参数是 Lambda 表达式，可将表达式写在圆括号外面
    list.associateTo(map){"疯狂${it}讲义" to it.length}
    // 输出{疯狂 Android 讲义=56, 疯狂 Java 讲义=4, 疯狂 Kotlin 讲义=6, 疯狂 Go 讲义=2}
    println(map)
    // 最后一个参数是 Lambda 表达式，可将表达式写在圆括号外面
    var rtx = list.reduce(){acc , e -> acc + e}
    println(rtx)
}
```

上面 3 行粗体字代码分别调用了 List 集合的 dropWhile()、associateTo()、reduce()方法，其最后一个参数都是 Lambda 表达式，因此程序在调用它们时可将 Lambda 表达式写在外面。

此外，如果 Lambda 表达式是函数调用的唯一参数，则调用方法时的圆括号完全可以省略。这意味着上面调用 dropWhile()、reduce()两个方法时，完全可以写成如下形式。

```
var rt = list.dropWhile {it.length > 3}
var rtx = list.reduce {acc , e -> acc + e}
```



提示：

根据上面介绍不难发现，通常我们建议将函数类型的形参放在形参列表的最后，这样方便以后传入 Lambda 表达式作为参数。

6.7.5 个数可变的参数和 Lambda 参数

前面介绍函数的个数可变的形参时提到：虽然 Kotlin 允许将个数可变的形参定义在形参列表的任意位置，但如果不将个数可变的形参放在形参列表的最后，那么就只能用命名参数的形式为可变形参之后的其他形参传入参数值。

但前一小节又建议将函数类型的形参放在形参列表的最后，此时就产生了一个问题：到底是将个数可变的形参放在最后，还是将函数类型的形参放在最后呢？

Kotlin 约定：如果调用函数时最后一个参数是 Lambda 表达式，则可将 Lambda 表达式放在圆括号外面，这样就无须使用命名参数了。因此答案是：如果一个函数既包含个数可变的形参，也包含函数类型的形参，那么就应该将函数类型的形参放在最后。例如如下程序。

程序清单：codes\06\6.7\TailLambda2.kt

```
fun <T> test(vararg names: String, transform: (String) -> T) : List<T>{
    var mutableList: MutableList<T> = mutableListOf()
    for (name in names) {
        mutableList.add(transform(name))
    }
    return mutableList.toList()
}
fun main(args: Array<String>) {
    // 将 Lambda 表达式放在圆括号后面，无须使用命名参数
    var list1 = test("Java", "Kotlin", "Go"){it.length}
    println(list1)
    // 将 Lambda 表达式放在圆括号后面，无须使用命名参数
    var list2 = test("Java", "Kotlin", "Go"){"疯狂${it}讲义"}
    println(list2)
}
```

上面程序中第一行粗体字代码定义了一个函数，该函数声明了两个形参，第一个是个数可变的形参（可传入多个参数值），第二个是函数类型的形参（可传入 Lambda 表达式）。

上面程序中的后面两行粗体字代码调用 test()方法时第二个参数是 Lambda 表达式，因此该参数可放在圆括号外面，而且无须使用命名参数。

6.8 匿名函数

Lambda 表达式虽然简洁、方便，但它有一个严重的缺陷：Lambda 表达式不能指定返回值类型。

大部分时候，由于 Kotlin 可以推断出 Lambda 表达式的返回值类型，因此即使不为 Lambda 表达式指定返回值类型也没有问题。但在一些特殊的场景下，如果 Kotlin 无法推断出 Lambda 表达式的返回值类型，此时就需要显式指定返回值类型，而匿名函数即可代替 Lambda 表达式。

6.8.1 匿名函数的用法

例如，下面代码示范了使用匿名函数来定义变量。

程序清单：codes\06\6.8\AnonymousFunTest.kt

```
fun main(args: Array<String>) {
    // 定义匿名函数，赋值给 test 变量
    var test = fun(x: Int, y: Int): Int {
        return x + y
    }
    // 通过 test 调用匿名函数
    println(test(2, 4))
}
```

上面粗体字代码定义了一个匿名函数。从上面代码可以看出，匿名函数与普通函数基本相似，只要将普通函数的函数名去掉就变成了匿名函数。

与普通函数不同的是，如果系统可以推断出匿名函数的形参类型，那么匿名函数允许省略形参类型。例如如下代码（程序清单同上）：

```
var filteredList = listOf(3, 5, 20, 100, -25).filter(
    // 使用匿名函数作为 filter()方法的参数
    fun(el): Boolean{
        return Math.abs(el) > 20
    })
println(filteredList)
```

上面程序调用 List 的 filter()方法时需要传入一个(Int)->Boolean 类型的参数，此时既可传入一个 Lambda 表达式，也可传入一个匿名函数，此处我们特意传入一个匿名函数作为参数。由于系统完全可以推断出该参数的类型必须是(Int)->Boolean，因此此处允许省略匿名函数的形参类型。

匿名函数的返回值类型的声明规则与普通函数相同：如果使用普通代码块作为函数体，则匿名函数需要显式指定返回值类型，否则认为该匿名函数没有返回值（相当于 Unit）；如果使用单表达式作为函数体，则无须指定返回值类型，系统可以自动推断出。

例如，如下代码为匿名函数定义单表达式作为函数体（程序清单同上）。

```
// 定义匿名函数的函数体是单表达式，可以省略声明函数的返回值类型
var wawa = fun(x: Int, y: Int) =  x + y
// 通过 wawa 调用匿名函数
println(wawa(2, 4))
var rt = listOf(3, 5, 20, 100, -25).filter(
    // 使用匿名函数作为 filter()方法的参数
    // 匿名函数的函数体是单表达式，允许省略声明函数的返回值类型
    fun(el) = Math.abs(el) > 20
)
println(rt)
```

上面第一行粗体字代码定义了一个匿名函数，该匿名函数的函数体是单表达式，因此此时可以省略声明函数的返回值类型（其实赋值给 wawa 的匿名函数与前面赋值给 test 的匿名函数是完全一样的，只是写法不同）；第二行粗体字代码再次定义了一个函数体为单表达式的匿名函数，同样省略了声明函数的返回值类型。

6.8.2 匿名函数和 Lambda 表达式的 return

匿名函数的本质依然是函数，因此匿名函数的 return 则用于返回该函数本身；而 Lambda 表达式的 return 用于返回它所在的函数，而不是返回 Lambda 表达式。例如如下代码。

程序清单：codes\06\6.8\ReturnVs.kt

```
fun main(args: Array<String>) {
    var list = listOf(3, 5, 30, -25, 14)
    // 使用匿名函数执行 forEach()方法
    list.forEach(fun(n) {
        println("元素依次为：${n}")
        // 匿名函数中的 return 用于返回该函数本身
        return
    })
```

```
    // 使用 Lambda 表达式执行 forEach()方法
    list.forEach({n ->
        println("元素依次为：${n}")
        // Lambda 表达式中的 return 用于返回它所在的函数（main 函数）
        return
    })
}
```

上面程序中分别使用了两种方式来调用 List 集合的 forEach()方法：第一种方式是使用匿名函数，匿名函数中的 return 用于返回该函数本身，因此对 main()函数没有任何影响，故这种方式可以正常遍历整个 List 集合；第二种方式是使用 Lambda 表达式，该表达式中的 return 用于返回它所在的函数（main 函数），因此该 forEach()方法只能输出一个数组元素，接下来就遇到结束整个 main 函数的 return 语句，从而导致程序结束——故第二种方式只能迭代一个集合元素。

注意：

上面代码之所以可以在 Lambda 表达式中直接使用 return，是因为该 forEach 方法使用了 inline 修饰，因此它相当于一个内联函数。关于内联函数的介绍可参考本章 6.10 节。



从上面程序中可以看出，匿名函数中的 return 语句用于返回该函数本身，但 Lambda 表达式中的 return 则用于返回该表达式所在的函数，因此一般不会在 Lambda 表达式中使用 return。如果真的很想过一下在 Lambda 表达式中使用 return 的瘾，则可使用限定返回的语法。例如，将上面第二段遍历改为如下形式（程序清单同上）：

```
// 使用 Lambda 表达式执行 forEach()方法
list.forEach({n ->
    println("元素依次为：${n}")
    // 使用限定返回，此时 return 只是返回传给 forEach 方法的 Lambda 表达式
    return@forEach
})
```

6.9 捕获上下文中的变量和常量

Lambda 表达式或匿名函数（以及局部函数、对象表达式）可以访问或修改其所在上下文（俗称“闭包”）中的变量和常量，这个过程被称为捕获。即使定义这些变量和常量的作用域已经不存在了，Lambda 表达式或匿名函数也依然可以访问或修改它们。

例如，如下程序先定义了一个函数，然后在该函数内定义了一个局部函数，此时局部函数就可以访问或修改其所在上下文（函数）中的变量。

程序清单：codes\06\6.9\CaptureTest.kt

```
// 定义一个函数，该函数的返回值类型为() -> List<String>
fun makeList(ele:String): () -> List<String> {
    // 创建一个不包含任何元素的 List
    var list: MutableList<String> = mutableListOf()
    fun addElement() : List<String> {
        // 向 list 集合中添加一个元素
```

```
        list.add(ele)
        return list
    }
    return ::addElement
}
```

上面程序中粗体字代码定义了一个局部函数，该局部函数即可访问或修改其所在函数中的变量。

上面程序中 makeList()函数的返回值类型为() -> List<String>，这表明该函数必须返回另一个函数，因此程序返回了粗体字代码定义的 addElement()局部函数，该函数负责向 list 集合添加元素。

在 makeList()函数中定义了一个 list 变量，该变量将作为 addElement()函数返回 List 集合；makeList()函数还定义了一个 ele 形参，该形参代表每次需要添加的集合元素。

下面分开来看程序中粗体字代码定义的局部函数。

```
fun addElement() : List<String> {
    // 向 list 集合中添加一个元素
    list.add(ele)
    return list
}
```

上面的 addElement()函数没有定义任何参数，也没有定义任何变量，但该局部函数却访问了 list 变量和 ele 形参，这是因为它捕获了其所在函数中的变量。

Lambda 表达式或匿名函数都会持有一个其所捕获的变量的副本，因此表面上看 addElement()访问的是 makeList()函数的 list 集合变量，但只要程序返回一个新的 addElement()函数，addElement()函数就会自己持有一个新的 list 的副本。为了证明这一点，为上面程序增加如下代码（程序清单同上）：

```
fun main(args: Array<String>){
    println("-----add1 返回的 List------")
    // add1 将会持有 list 的副本
    val add1 = makeList("孙悟空")  // ①
    println(add1())
    println(add1())
    println("-----add2 返回的 List------")
    // add2 将会持有 list 的副本，与 add1 的 list 副本没有关系
    val add2 = makeList("猪八戒")  // ②
    println(add2())
    println(add2())
}
```

上面程序中①号粗体字代码调用 makeList()返回一个 addElement()局部函数的实例，而 add1 变量则引用该局部函数的实例。add1 局部函数拥有其所在封闭函数（makeList）中 list 的副本，这样保证程序多次调用 add1 时，list 总引用 add1()所返回的 List 集合。

②号粗体字代码调用 makeList()再次返回一个 addElement()局部函数的实例，而 add2 变量依然引用该局部函数的实例。add2 局部函数同样拥有一个 list 的副本，add2 所拥有的 list 副本与 add1 所拥有的 list 副本没有任何关系。

编译、运行该程序，即可看到如下输出：

```
-----add1 返回的 List------
[孙悟空]
```

```
[孙悟空, 孙悟空]
-----add2 返回的 List------
[猪八戒]
[猪八戒, 猪八戒]
```

6.10 内联函数

先简单介绍一下高阶函数（为函数传入函数或 Lambda 表达式作为参数）的调用过程。调用 Lambda 表达式或函数的过程是：程序要将执行顺序转移到被调用表达式或函数所在的内存地址，当被调用表达式或函数执行完后，再返回到原函数执行的地方。

在上面这个转移过程中，系统要处理如下事情。

- 为被调用的表达式或函数创建一个对象。
- 为被调用的表达式或函数所捕获的变量创建一个副本。
- 在跳转到被调用的表达式或函数所在的地址之前，要先保护现场并记录执行地址；从被调用的表达式或函数地址返回时，要先恢复现场，并按原来保存的地址继续执行。也就是通常所说的压栈和出栈。

从上面介绍不难看出，函数调用会产生一定的时间和空间开销，如果被调用的表达式或函数的代码量本身不大，而且该表达式或函数经常被调用，那么这个时间和空间开销的损耗就很不划算。

为了避免产生函数调用的过程，我们可以考虑直接把被调用的表达式或函数的代码“嵌入”原来的执行流中——简单来说，就是编译器负责去“复制、粘贴”：复制被调用的表达式或函数的代码，然后粘贴到原来的执行代码中。为了让编译器帮我们干这个复制、粘贴的活，可通过内联函数来实现。

6.10.1 内联函数的使用

使用内联函数非常简单，只要使用 inline 关键字修饰带函数形参的函数即可。下面程序示范了内联函数和非内联函数的区别。

程序清单：codes\06\6.10\InlineTest.kt

```
// 定义函数类型的形参，其中 fn 是(Int) -> Int 类型的形参
inline fun map(data : Array<Int>, fn: (Int) -> Int) : Array<Int> {
    var result = Array<Int>(data.size, {0})
    // 遍历 data 数组的每个元素，并用 fn 函数对 data[i]进行计算
    // 然后将计算结果作为新数组的元素
    for (i in data.indices) {
        result[i] = fn(data[i])
    }
    return result
}
fun main(args: Array<String>) {
    var arr = arrayOf(20, 4, 40, 100, 30)
    var mappedResult1 = map(arr, {it + 3})
    println(mappedResult1.contentToString())
}
```

上面 map()函数包含一个函数类型的形参，且该函数使用了 inline 修饰，因此它是一个内

联函数。编译该程序，会发现编译结果只产生一个 InlineTestKt.class 文件，不会生成其他额外的内部类的 class 文件，这表明编译器实际上会将 Lambda 表达式的代码复制、粘贴到 map() 函数中。

也就是说，程序调用的 map 函数编译之后实际上变成了如下形式：

```
fun map(data : Array<Int>, fn: (Int) -> Int) : Array<Int> {
    var result = Array<Int>(data.size, {0})
    // 遍历 data 数组的每个元素，并用 fn 函数对 data[i]进行计算
    // 然后将计算结果作为新数组的元素
    for (i in data.indices) {
        result[i] = data[i] + 3
    }
    return result
}
```

从上面的粗体字代码可以看出，此时根本就不存在函数调用，自然也就不需要额外生成函数对象了，也不会产生捕获，也不需要处理函数调用的压栈和出栈开销。

如果去掉 InlineTest.kt 程序中 map 函数的 inline 修饰符，再次编译该函数，将会看到系统生成 InlineTestKt.class 和 InlineTestKt$main$mappedResult1$1.class 文件，这表明系统将会为 Lambda 表达式额外生成一个函数对象，自然也会产生函数调用的压栈和出栈开销。

介绍到此处，可能有读者会想，既然内联函数这么好，那么我们干脆全部使用内联函数好了。下面的问题是：内联函数的缺点在哪里呢？到底哪些情况应该使用内联函数，哪些情况不应该使用内联函数呢？

通过前面的介绍我们知道，内联函数的本质是将被调用的 Lambda 表达式或函数的代码复制、粘贴到原来的执行函数中，因此如果被调用的 Lambda 表达式或函数的代码量非常大，且该 Lambda 表达式或函数多次被调用——注意每调用一次，该 Lambda 表达式或函数的代码就会被复制一次，因此势必带来程序代码量的急剧增加。

由此可见，内联函数并不总是带来好处的，内联函数是以目标代码的增加为代价来节省时间开销的。因此是否应该使用内联函数的答案是：如果被调用的 Lambda 表达式或函数包含大量的执行代码，那么就不应该使用内联函数；如果被调用的 Lambda 表达式或函数只包含非常简单的执行代码（尤其是单表达式），那么就应该使用内联函数。

6.10.2 部分禁止内联

使用 inline 修饰函数之后，所有传入该函数的 Lambda 表达式或函数都会被内联化；如果我们又希望该函数中某一个或某几个函数类型的形参不会被内联化，则可考虑使用 noinline 修饰它们。

简单来说，noinline 用于显式阻止某一个或某几个形参内联化。

例如如下程序。

程序清单：codes\06\6.10\NoInlineTest.kt

```
inline fun test(fn1: (Int)->Int, noinline fn2:(String)->String){
    println(fn1(20))
    println(fn2("Kotlin"))
}
fun main(args: Array<String>) {
    test({it * it}, {"疯狂${it}讲义"})
}
```

上面程序中的粗体字代码使用 inline 修饰了 test()函数，该函数带有两个函数类型的形参，因此在默认情况下，所有传入 test()函数的 Lambda 表达式或函数都会被内联化——如果没有为后面第二个 fn2 形参使用 noinline 修饰，编译该程序将只会看到一个 NoInlineTest.class 文件，这意味着传给该函数的两个表达式都被直接复制到 test()函数中。

如果为 test()函数的 fn2 形参添加了 noinline 修饰符，再次编译该文件，将会看到系统生成 NoInlineTestKt.class 和 NoInlineTestKt$main$1.class 两个文件，这就意味着 Kotlin 取消了调用 test()函数所传入的第二个 Lambda 表达式的内联化，因此编译时只需为传入 test()函数的第一个 Lambda 表达式额外生成一个函数对象。



提示：

如果彻底去掉 test()函数前面的 inline 修饰符，编译该文件将会看到生成 3 个 class 文件，即 NoInlineTestKt.class、NoInlineTestKt$main$1.class 和 NoInlineTestKt-$main$2.class，这表明系统为传给 test()函数的两个 Lambda 表达式都会额外生成一个函数对象。

6.10.3 非局部返回

前面已经提到，在 Lambda 表达式中直接使用 return 不是用于返回该表达式，而是用于返回该表达式所在的函数。但要记住：在默认情况下，在 Lambda 表达式中并不允许直接使用 return。这是因为如果是非内联的 Lambda 表达式，该 Lambda 表达式会额外生成一个函数对象，因此这种表达式中的 return 不可能用于返回它所在的函数。

由于内联的 Lambda 表达式会被直接复制、粘贴到调用它的函数中，故此时在该 Lambda 表达式中可以使用 return，该 return 就像直接写在 Lambda 表达式的调用函数中一样。因此，该内联的 Lambda 表达式中的 return 可用于返回它所在的函数，这种返回被称为非局部返回。

例如如下代码。

程序清单：codes\06\6.10\NonLocalReturn.kt

```
// 定义函数类型的形参，其中 fn 是(Int) -> Unit 类型的形参
inline fun each(data : Array<Int>, fn: (Int)->Unit) {
    for (el in data) {
        fn(el)
    }
}
fun main(args: Array<String>) {
    var arr = arrayOf(20, 4, 40, 100, 30)
    each(arr, {
        println(it)
        return // 如果 each 函数没有 inline 修饰，此处编译错误
        // 如果 each 函数有 inline 修饰，此处的 return 将返回 main 函数
    })
}
```

上面程序中的粗体字代码在 Lambda 表达式中使用了 return 语句，如果该 Lambda 表达式出现在 inline 函数中，那么该 Lambda 表达式中的 return 用于返回它所在的函数(main()函数)，因此会看到上面程序只能输出一个数组元素，程序就因为 return 退出了。

如果删除上面 each()函数中的 inline 修饰符，那么下面的 Lambda 表达式中的 return 将会

提示编译错误：'return' is not allowed here，这意味着在非内联的 Lambda 表达式中不能使用 return。

另外，某些内联函数不是直接从函数体中调用 Lambda 表达式的，而是从其他的执行上下文（如局部对象或局部函数）中来获取 Lambda 表达式的。在这种情况下，非局部返回的控制流也不允许出现在 Lambda 表达式中。此时应该使用 crossinline 来修饰该参数。例如如下代码片段：

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```



提示：

目前内联的 Lambda 表达式目前暂不支持 break 和 continue 流程控制，但未来 Kotlin 计划支持该功能。

6.11 本章小结

本章所介绍的函数和 Lambda 表达式是 Kotlin 编程的两大核心机制之一，Kotlin 语言既支持面向过程编程，也支持面向对象编程。而函数和 Lambda 表达式就是 Kotlin 面向过程编程的语法基础，而且 Kotlin 的函数和 Lambda 表达式的功能非常强大，因此读者必须加以重视。

学习本章内容不仅需要掌握定义函数、调用函数的语法，还需要掌握函数形参的外部形参名、形参默认值、常量形参和变量形参、In-Out 形参等高级特性。除此之外，函数类型还可以被当成数组类型使用，既可用于声明变量，也可作为其他函数的形参类型和返回值类型——简而言之，函数类型就是引用类型。

本章的另一个重点是 Lambda 表达式。Lambda 表达式相当于简化的匿名函数，读者不仅需要掌握 Lambda 表达式的基本语法，还需要掌握其各种简化形式。另外，还需要注意 Lambda 表达式会捕获上下文中的变量。

CHAPTER

7

第7章 面向对象（上）

本章要点

- 定义类、属性和方法
- 创建并使用对象
- 方法与函数的关系
- 通过中缀函数实现自定义运算符
- componentN 方法与解构
- 定义只读属性和读写属性
- 幕后字段和幕后属性
- Kotlin 的包和导包
- 使用访问控制符实现封装
- 构造器的作用和构造器重载
- 主构造器和次构造器
- 继承的语法
- 子类构造器委托调用父类构造器
- 重写父类的方法和属性
- super 关键字的用法
- 继承和多态
- 向上转型和强制类型转换

Kotlin 支持面向对象编程，Kotlin 提供了定义类、属性、方法等最基本的功能。类可被认为是一种自定义的数据类型，可以使用类来定义变量，所有使用类定义的变量都是引用变量，它们将会引用类的对象。类用于描述客观世界中某一类对象的共同特征，而对象则是类的具体存在，Kotlin 也使用构造器来创建类的对象。

Kotlin 也支持面向对象的三大特征：封装、继承和多态。Kotlin 提供了 private、protected、internel 和 public 四个访问控制符来实现良好的封装；子类继承父类就可以继承父类的属性和方法，如果访问权限允许，子类对象可以直接调用父类中定义的方法。

7.1 类和对象

类是面向对象的重要内容，可以把类当成一种自定义类型，可以使用类来定义变量，这种类型的变量统称为引用变量。也就是说，所有类都是引用类型。

7.1.1 定义类

在面向对象的程序设计过程中有两个重要概念：类（class）和对象（object，也被称为实例，instance）。其中类是某一批对象的抽象，可以把类理解成某种概念；对象才是一个具体存在的实体。从这个意义上看，比如日常所说的人，其实都是人的对象，而不是人类。

Kotlin 定义类的简单语法如下：

```
[修饰符] class 类名 [constructor 主构造器]{
    零个到多个次构造器定义...
    零个到多个属性...
    零个到多个方法...
}
```

注意：

Kotlin 的官方文档很混乱，它一会将方法称为函数（function），一会又称为方法（method）。本书严格按照行业约定：独立的才叫函数，在类中定义的叫方法。

在上面的语法格式中，修饰符可以是 public | internal | private（只能出现其中之一）、final | open | abstract（也只能出现其中之一），或者完全省略修饰符。类名只要是一个合法的标识符即可，但这仅仅满足的是 Kotlin 的语法要求；如果从程序的可读性方面来看，Kotlin 的类名必须是由一个或多个有意义的单词连缀而成的，每个单词首字母大写，其他字母全部小写，单词与单词之间不要使用任何分隔符。

提示：

如果将 open 和 abstract 两个修饰符放在一起使用，编译器会发出警告。open 是 Kotlin 新引入的修饰符，open 是 final 的反义词，用于修饰一个类、方法或属性，表明类可派生子类，方法或属性可被重写。

Kotlin 的类定义由类名、类头（指定其泛型声明、主构造器等）和用花括号包围的类体构成。类头和类体都是可选的。

对于一个类定义而言，可以包含三种最常见的成员：构造器、属性和方法，这三种成员都

可以定义零个或多个，如果三种成员都只定义零个，就是定义了一个空类。空类没有类体，可以省略花括号。例如，如下类定义是允许的。

```
class Empty
```

通常来说，空类没有太大的实际意义。

类中各成员之间的定义顺序不会产生任何影响，各成员之间可以相互调用。

属性用于定义该类的对象所包含的状态数据，方法则用于定义该类的对象的行为特征或者功能实现。构造器用于构造该类的对象，Kotlin通过调用构造返回该类的对象（无须使用new）。

构造器是一个类创建对象的根本途径，如果一个类没有构造器，这个类通常无法创建对象。一个 Kotlin 类可以有 0~1 个主构造器和 0~*N* 个次构造器。主构造器是类头的一部分，它跟在类名（和泛型声明）后。

```
class User constructor(firstName: String) {
}
```

上面的粗体字代码就是主构造器（主构造器属于类头部分），由此可见，主构造器就是在类头使用 constructor 关键字定义一个无执行体的构造器。虽然主构造器不能定义执行体，但可以定义多个形参，这些形参可以在属性声明、初始化块中使用。

如果主构造器没有任何注解或修饰符，则可以省略 constructor 关键字。即上面代码可改为如下形式：

```
class User(firstName: String) {
}
```

Kotlin 还提供了一个功能：如果程序员没有为非抽象类定义任何（主或次）构造器，系统会自动提供一个无参数的主构造器，该构造器默认用 public 修饰。一旦程序员为一个类提供了构造器，系统将不再为该类提供构造器。

定义属性的语法格式如下：

```
[修饰符] var|val 属性名: 类型 [= 默认值]
    [<getter>]
    [<setter>]
```

对定义属性的语法格式详细说明如下。

- **修饰符**：修饰符可以省略，也可以是 public | protected | internal | private、final | open | abstruct，public | protected | private 只能出现其中之一，final | open | abstract 也只能出现其中之一。对于属性使用 public、protected、internal 修饰和不使用访问控制符，效果是一样的。如果使用 private 修饰该属性，该属性将作为幕后属性使用。



提示：

关于幕后属性的介绍，请参考 7.3 节。

- **var|val**：使用 var 声明读写属性，使用 val 声明只读属性（不能修改）。
- **属性名**：从语法的角度来说，属性名只要是一个合法的标识符即可；从程序可读性的角度来看，属性名应该由一个或多个有意义的单词连缀而成，第一个单词首字母小写，后面每个单词首字母大写，其他字母全部小写，单词与单词之间不要使用任何分隔符。属性用于描述一个类的对象包含的状态数据，因此建议属性名使用英文名词。

- **类型**：类型可以是 Kotlin 允许的任何数据类型。如果 Kotlin 可以从初始值或 getter 方法的返回值推断出属性的类型，那么此处可省略属性的类型。
- **默认值**：定义属性时还可以指定一个可选的默认值。要么在此处指定初始值，要么在构造器或初始化块中指定初始值。
- **getter、setter**：用于为该属性编写自定义的 getter、setter 方法。如果不指定，Kotlin 会为读写属性提供默认的 getter、setter 方法；为只读属性提供默认的 getter 方法（只读属性不允许定义 setter 方法）。

定义方法的语法和前面定义函数的语法完全一样，换句话说，如果将函数放在类中定义，那么它就变成了方法，故此处不再详述。

顶层函数的语法与类中方法的语法的主要区别就在于修饰符：顶层函数不能使用 protected、abstract 和 final 修饰符，但类中的方法可使用 public | protected | internal | private、final | abstract | open 这些修饰符。

定义构造器的语法格式如下：

```
[修饰符] constructor (形参列表){
    // 由零条到多条可执行语句组成的构造器执行体
}
```

对定义构造器的语法格式详细说明如下。

- **修饰符**：修饰符可以省略，也可以是 public、protected、internal、private 其中之一。
- **形参列表**：其格式和定义方法的形参列表的格式完全相同。

值得指出的是，构造器既不能定义返回值类型，也不能使用 Unit 声明构造器没有返回值。

下面程序将定义一个 Person 类。

程序清单：codes\07\7.1\Person.kt

```
class Person {
    // 下面定义了两个属性
    var name: String = ""
    var age: Int = 0
    // 下面定义了一个 say 方法
    fun say(content: String){
        println(content)
    }
}
```

在上面的 Person 类中没有定义构造器，系统将为它提供一个默认的构造器，系统提供的构造器总是没有参数的。

定义类之后，接下来即可使用该类了。Kotlin 的类大致有如下作用。

- 定义变量。
- 创建对象。
- 派生子类。

下面先介绍使用类来定义变量和创建对象。

7.1.2 对象的产生和使用

创建对象的根本途径是构造器，调用某个类的构造器即可创建这个类的对象，并且无须使

用 new 关键字。

程序清单：codes\07\7.1\PersonTest.kt

```
// 使用 Person 类定义一个 Person 类型的变量
var p: Person
// 调用 Person 类的构造器，返回一个 Person 对象
// 将该 Person 对象赋给 p 变量
p = Person()
```



提示：

如果在命令行使用 kotlinc 编译上面程序，则需要为 kotlinc 添加-cp .选项，-cp 用于告诉 kotlinc 编译时需要从哪些路径加载类，该选项后的点指定从当前路径加载类。

上面代码也可简写成如下形式：

```
// 在定义 p 变量的同时为 p 变量赋值
var p: Person = Person()
```

创建对象之后，接下来即可使用该对象了。Kotlin 的对象大致有如下作用。

- 访问对象的属性。
- 调用对象的方法。

如果访问权限允许，那么在类中定义的方法和属性都可通过对象来调用。通过对象访问方法或属性的语法是：对象.属性|方法(参数)。在这种方式中，对象是主调者，用于访问该对象的属性或方法。

下面代码通过 Person 对象来调用 Person 的属性和方法（程序清单同上）。

```
// 访问 p 的 name 属性，直接为该属性赋值
p.name = "李刚"
// 调用 p 的 say()方法，声明 say()方法时定义了一个形参
// 调用该方法必须为形参指定一个值
p.say("Kotlin 语言很简单，学习很容易！")
// 直接输出 p 的 name 属性，将输出“李刚”
println(p.name)
```

上面代码中通过 Person 对象调用了 say()方法，调用方法时必须为方法的形参赋值。因此在这行代码中调用 Person 对象的 say()方法时，必须为 say()方法传入一个字符串作为形参的参数值，这个字符串将被赋给 content 参数。

大部分时候，定义一个类就是为了重复创建该类的对象，同一个类的多个对象具有相同的特征，而类则定义了多个对象的共同特征。从某个角度来看，类定义的是多个对象的特征，因此类不是一种具体存在，对象才是具体存在。比如完全可以这样说：你不是人这个类，我也不是人这个类，我们都只是人的对象。

7.1.3 对象的 this 引用

Kotlin 也提供了 this 关键字，this 关键字总是指向调用该方法的对象。根据 this 出现位置的不同，this 作为对象的默认引用有两种情形。

- 在构造器中引用该构造器正在初始化的对象。
- 在方法中引用调用该方法的对象。

在后面介绍的扩展函数或带接收者的匿名函数中，this 代表点号左侧传递的接收者。

this 关键字最大的作用就是让类中的一个方法访问该类的另一个方法或属性。假设定义了一个 Dog 类，这个 Dog 对象的 run()方法需要调用它的 jump()方法，此时就可通过 this 关键字作为 jump()方法的调用者。

this 可以代表任何对象，当 this 出现在某个方法体中时，它所代表的对象是不确定的，但它的类型是确定的——它所代表的只能是当前类的实例；只有当这个方法被调用时，它所代表的对象才被确定下来——谁在调用这个方法，this 就代表谁。

例如，定义如下 Dog 类。

程序清单：codes\07\7.1\Dog.kt

```
class Dog
{
    // 定义一个 jump()方法
    fun jump()
    {
        println("正在执行 jump 方法")
    }
    // 定义一个 run()方法，run()方法需要借助 jump()方法
    fun run()
    {
        // 使用 this 引用调用 run()方法的对象
        this.jump()
        println("正在执行 run 方法")
    }
}
```

上面 run()方法中的 this 代表了该方法的调用者：谁在调用 run()方法，那么 this 就代表谁。因此该方法表示：当一个 Dog 对象调用 run()方法时，run()方法需要依赖它自己的 jump()方法。

在现实世界里，对象的一个方法依赖于另一个方法的情形很常见，例如，吃饭方法依赖于拿筷子方法，写程序方法依赖于敲键盘方法……这种依赖都是同一个对象的两个方法之间的依赖。因此，Kotlin 允许对象的一个成员直接调用另一个成员，可以省略 this 前缀。也就是说，将上面的 run()方法改为如下形式也完全正确。

```
// 定义一个 run()方法，run()方法需要借助 jump()方法
fun run()
{
    // 使用 this 引用调用 run()方法的对象
    jump()
    println("正在执行 run 方法")
}
```

大部分时候，一个方法访问该类中定义的其他方法、属性时加不加 this 前缀，效果是完全一样的。



提示：

省略 this 前缀只是一种假象，虽然程序员省略了调用 jump()方法之前的 this，但实际上这个 this 依然存在。根据汉语语法习惯，完整的语句至少包括主语、谓语、宾语，在面向对象的世界里，主、谓、宾的结构完全成立，例如“猪八戒吃西瓜”是一条汉语语句，转换为面向对象的语法，就可以写成“猪八戒.吃(西瓜)”，因此

本书常常把调用属性、方法的对象称为“主调（主语调用者的简称）”。对于 Kotlin 来说，调用属性、方法时，主调是必不可少的，即使代码中省略了主调，但实际的主调依然存在。一般来说，如果调用属性、方法时省略了前面的主调，那么默认使用 this 作为主调。

大部分时候，一个方法调用类中的其他方法、属性时无须使用 this 前缀，但如果方法中有一个局部变量和属性同名，但程序又需要在该方法中访问这个被隐藏的属性，则必须使用 this 前缀。关于局部变量覆盖属性的情形，请参见 7.3 节的内容。

除此之外，this 引用也可用于构造器中作为默认引用。由于构造器是直接调用的，而不是使用对象来调用的，所以 this 在构造器中代表该构造器正在初始化的对象。

程序清单：codes\07\7.1\ThisInConstructor.kt

```
class ThisInConstructor{
    // 定义一个名为 foo 的属性
    var foo: Int
    constructor(){
        // 在构造器中定义一个 foo 变量
        val foo = 0
        // 使用 this 代表该构造器正在初始化的对象
        // 下面的代码将会把该构造器正在初始化的对象的 foo 属性设为 6
        this.foo = 6
    }
}
fun main(args: Array<String>){
    // 所有使用 ThisInConstructor 创建的对象的 foo 属性将被设为 6
    println(ThisInConstructor().foo) // 输出 6
}
```

在 ThisInConstructor 构造器中使用 this 引用时，this 总是引用该构造器正在初始化的对象。程序中粗体字代码将正在执行初始化的 ThisInConstructor 对象的 foo 属性设为 6，这意味着该构造器返回的所有对象的 foo 属性都等于 6。

与普通方法类似的是，大部分时候，在构造器中访问其他属性和方法时都可以省略 this 前缀，但如果构造器中有一个与属性同名的局部变量，又必须在构造器中访问这个被覆盖的属性，则必须使用 this 前缀，如上面的 ThisInConstructor.kt 所示。

当 this 作为对象的默认引用使用时，程序可以像访问普通变量一样来访问这个 this 引用，甚至可以把 this 当成普通方法的返回值。看下面程序。

程序清单：codes\07\7.1\ReturnThis.kt

```
class ReturnThis {
    var age = 0
    fun grow(): ReturnThis{
        age++
        // return this 返回调用该方法的对象
        return this
    }
}
fun main(args: Array<String>) {
    val rt = ReturnThis()
    // 可以连续调用同一个方法
```

```
    rt.grow()
        .grow()
        .grow()
    println("rt 的 age 属性值是:" + rt.age)
}
```

从上面程序中可以看出，如果在某个方法中把 this 作为返回值，则可以多次连续调用同一个方法，从而使得代码更加简洁。但是，这种把 this 作为返回值的方法可能造成实际意义的模糊，例如上面的 grow 方法，用于表示对象的生长，即 age 属性的值加 1，实际上不应该有返回值。



注意：

使用 this 作为方法的返回值可以让代码更加简洁，但可能造成实际意义的模糊。



7.2 方法详解

方法是类或对象的行为特征的抽象，方法是类或对象最重要的组成部分。但 Kotlin 的方法并不仅仅是单纯的方法，Kotlin 的方法与函数也有极大的关关系。

7.2.1 方法与函数的关系

Kotlin 的方法与函数其实是统一的，不仅定义函数和方法的语法相同，而且定义在类中的方法依然可独立出来。也就是说，即使我们将方法定义在类里面，这个方法也依然可以转换为函数。

例如如下代码。

程序清单：codes\07\7.2\MethodTest.kt

```
class Dog{
    // 定义一个无参数的 run 方法
    // 其类型是 ()->Unit
    fun run(){
        println("run 方法")
    }
    // 定义一个带 String 参数的 run 方法
    // 其类型是 (String)->Unit
    fun eat(food: String){
        println("正在吃: " + food)
    }
}
fun main(args: Array<String>) {
    // 将 Dog 的 run 方法赋值给 rn 变量
    // rn 变量的类型应该是 (Dog)->Unit
    var rn:(Dog)->Unit = Dog::run
    val d = Dog()
    rn(d)
    // 将 Dog 的 eat 方法赋值给 et 变量
    // et 变量的类型应该是 (Dog, String)->Unit
    var et = Dog::eat
    et(d, "肉骨头")
}
```

上面程序定义了一个 Dog 类，并为该 Dog 类定义了两个方法，但这两个方法依然可以被分离出来使用。接下来程序的 main()函数中的两行粗体字代码负责将 run()、eat()方法依次赋值给 rn、et 变量。

为了与前面介绍的获取函数的引用保持一致，这里要获取某个方法的引用同样需要在方法名前面添加两个冒号（::）。由于此处的方法属于特定的类，因此还需要在方法引用之前添加类名。

对于第一行粗体字代码：var rn:(Dog)->Unit = Dog::run，我们显式指定了 rn 变量的类型，虽然 Dog 的 run()方法是()->Unit 类型，但当程序将 run()方法独立成函数时，调用 run()方法的调用者（Dog 对象）将作为第一个参数传入，因此 Dog::run 的实际类型是(Dog)->Unit。

对于第二行粗体字代码：var et = Dog::eat，我们没有指定 et 变量的类型，但由于程序将 Dog::eat 赋值给 et 变量，系统将自动推断出 et 的类型是(Dog, String)->Unit。

7.2.2 中缀表示法

Kotlin 的方法还可使用 infix 修饰，这样该方法就可通过中缀表示法调用，就像这些方法是双目运算符一样。需要指出的是，infix 方法只能有一个参数——原因很简单，因为双目运算符的后面只能带一个参数。



由于双目运算符的后面只能带一个参数，因此 infix 方法只能带一个参数。



下面程序定义了一个简单的 Apple 类，并为该类定义了两个 infix 方法，并且使用双目运算符的方式来调用这两个方法。

程序清单：codes\07\7.2\InfixTest.kt

```
class ApplePack(weight: Double){
    var weight = weight
    override fun toString(): String{
        return "ApplePack[weight=${this.weight}]"
    }
}
class Apple(weight: Double){
    var weight = weight
    override fun toString(): String{
        return "Apple[weight=${this.weight}]"
    }
    // 定义中缀方法，使用 infix 修饰
    infix fun add(other: Apple): ApplePack{
        return ApplePack(this.weight + other.weight)
    }
    // 定义中缀方法，使用 infix 修饰
    infix fun drop(other: Apple): Apple{
        this.weight -= other.weight
        return this
    }
}
fun main(args: Array<String>) {
    var origin = Apple(3.4)
```

```
    // 使用 add 方法
    val ap = origin add Apple(2.4)
    println(ap)
    origin drop Apple(1.4)
    println(origin)
}
```

上面程序中前两行粗体字代码定义了两个 infix 方法，其实没有任何特别的地方，只要注意两点：为方法添加 infix 修饰；该方法只有一个形参。

接下来 main()函数中的两行粗体字代码示范了使用双目运算符的语法调用 infix 方法——两个 Apple 执行 add 运算（调用 add()方法）时，程序返回了一个 ApplePack 实例；两个 Apple 执行 drop 运算（执行 drop()方法）时，程序将名为 origin 的 Apple 对象的重量减少了。



提示：

上面程序为 Apple、ApplePack 重写了 toString()方法，Kotlin 类的 toString()方法的作用与 Java 类的 toString()方法的作用相同。

7.2.3 componentN 方法与解构

Kotlin 允许将一个对象的 *N* 个属性“解构”给多个变量，写法如下：

```
var (name, pass) = user
```

上面这行代码相当于将 user 对象的两个属性分别赋值给 name、pass 两个变量，这两个变量的类型会根据 user 对象的属性类型来推断。

看到这里，可能有读者蒙了：Kotlin 怎么知道把哪两个属性分别赋值给 name、pass 变量呢？其实 Kotlin 会将上面的赋值代码转换为如下两行：

```
var name = user.component1()
var pass = user.component2()
```

从上面介绍可以看出，如果希望将对象解构给多个变量，那么必须为该对象的类定义 componentN()方法。程序希望将对象解构给几个变量，就需要为该类定义几个 componentN()方法，且该方法需要使用 operator 修饰。

下面程序示范了为 User 类定义 componentN()方法，然后即可将该对象解构给 3 个变量。

程序清单：codes\07\7.2\DestructTest.kt

```
class User(name: String, pass: String, age: Int){
    var name = name
    var pass = pass
    var age = age
    // 定义 operator 修饰的 componentN 方法，用于解构
    operator fun component1(): String{
        return this.name
    }
    // 定义 operator 修饰的 componentN 方法，用于解构
    operator fun component2(): String{
        return this.pass
    }
    // 定义 operator 修饰的 componentN 方法，用于解构
    operator fun component3(): Int{
        return this.age
```

```
    }
}
fun main(args: Array<String>) {
    // 创建 User 对象
    val user = User("yeeku", "668899", 29)
    // 将 User 对象解构给 2 个变量
    // 只利用 user 对象的 component1()和 component2()方法
    var (name, pass: String) = user
    println(name)
    println(pass)
    // 将 User 对象解构给 3 个变量
    // 利用 user 对象的 component1()、component2()和 component3()方法
    var (name2, pass2, age) = user
    println(name2)
    println(pass2)
    println(age)
}
```

上面程序中前 3 行粗体字代码为 User 类定义了 3 个 operator 修饰的 compoentN()方法，这就表明该类的实例最多可同时解构给 3 个变量。

接下来 main()函数中的第一行粗体字代码将 user 对象解构给 2 个变量，此时将只解构 user 对象的前两个 componentN()方法的返回值。这行代码还为 pass 变量声明了类型：String，实际上该类型是可推断出来的，因此完全可以不指定。

第二行粗体字代码将 user 对象解构给 3 个变量，此时将会解构 user 对象的前 3 个 componentN()方法的返回值。

在某些时候，程序希望解构对象后面几个 componentN()方法的返回值、忽略前面几个 componentN()方法的返回值，此时可通过下画线（_）来占位。例如如下代码（程序清单同上）：

```
// 如果不想要前面的某个属性，用“_”代替它
var (_, pass3, age2) = user
println(pass3)
println(age2)
```

上面粗体字代码希望解构 user 对象的 component2()、component3()两个方法的返回值，忽略 component1()方法的返回值，因此程序在圆括号中的第一个位置处使用了下画线来代替。

理解了对象解构的用法之后，接下来详细谈谈遍历 Map 的语法。我们知道可使用如下语法来遍历 Map。

```
for ((key, value) in map) {
    // 使用 key、value
}
```

程序可通过 for-in 循环来遍历 Map 结构，这是由于 Map 包含了一个 operator 修饰的 iterator()方法，且该方法返回了 Iterator<Map.Entry<K, V>>对象。

Iterator 代表了可迭代的对象，它的泛型参数代表了被迭代的元素的类型——Map.Entry<K, V>（Entry 是 Map 的嵌套类），而 Map.Entry 则定义了如下两个方法：

```
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

上面两个方法就是 Map.Entry 提供的 operator 修饰的 componentN()方法，因此程序可执行如下解构：

```
var (key, value) = Map.Entry 对象
```

Map 本身是可迭代的，我们原本可按如下方式遍历 Map 集合。

```
for (entry in map) {
    // 使用 entry 获取 key、value
}
```

但是由于 Kotlin 允许对 entry 执行解构操作，因此上面的遍历可改写为如下形式：

```
for ((key, value) in map) {
    // 使用 entry 获取 key、value
}
```

7.2.4 数据类和返回多个值的函数

Kotlin 本身并不支持定义返回多个值的函数或方法，但通过上面所介绍的对象解构，我们同样可让 Kotlin 函数返回多个值——本质是让 Kotlin 返回一个支持解构的对象。

为了简化解构的实现，Kotlin 提供了一种特殊的类：数据类。数据类专门用于封装数据。



提示：

如果读者有实际的 Java 开发经验，一定不会忘记在 Java 中开发的 DTO、VO 等，这些类基本不包含任何逻辑，它们只是用于封装数据，因此开发者只要为它们定义 field，并为每个 field 提供 setter、getter 方法即可。而 Kotlin 提供的数据类正是用于简化 DTO、VO 的开发的。

数据类除使用 data 修饰之外，还要满足如下要求。

- 主构造器至少需要有一个参数。
- 主构造器的所有参数需要用 val 或 var 声明为属性。
- 数据类不能用 abstract、open、sealed 修饰，也不能定义成内部类。
- 在 Kotlin 1.1 之前，数据类只能实现接口；现在数据类也可继承其他类。

定义数据类之后，系统自动为数据类生成如下内容。

- 生成 equals()/hashCode()方法。
- 自动重写 toString()方法，返回形如"User(name=John, age=42)"的字符串。
- 为每个属性自动生成 operator 修饰的 componentN()方法。
- 生成 copy()方法，用于完成对象复制。

下面程序将会先定义一个数据类，然后通过数据类来实现返回多个值的函数。

程序清单：codes\07\7.2\MultiReturn.kt

```
// 定义一个数据类
// 数据类会自动为每个属性定义对应的 componentN 方法
data class Result(val result: Int, val status: String)
fun factorial(n: Int): Result{
    if(n == 1){
        return Result(1, "成功")
    } else if(n > 1) {
        return Result(factorial(n - 1).result * n, "成功")
    } else {
        return Result(-1, "参数必须大于 0")
    }
}
```

```
fun main(args: Array<String>) {
    // 通过解构获取函数返回的两个值
    var(rt, status) = factorial(6)
    println(rt)
    println(status)
}
```

上面程序中定义了一个 factorial()函数，该函数的返回值类型是 Result 数据类，该数据类会自动为它的 result、status 两个属性生成 componentN()方法，因此 Result 对象支持解构。

上面程序的 main()函数中的粗体字代码将 factorial()函数的执行结果解构给 rt、status 两个变量，这就相当于让该函数返回了两个值。

与前面类似的是，如果程序只需要解构 component2()方法的返回值，则可用“_”忽略 component1()方法的返回值。代码如下（程序清单同上）：

```
var(_, status2) = factorial(-6)
println(status2)
```

系统为数据类生成的 copy()方法可用于复制对象，例如如下代码（程序清单同上）：

```
var result = Result(2, "未知结果")
// 调用 copy()方法完成复制
val oldRt = result.copy()
println(oldRt)
```



提示：

Kotlin 标准库提供了 Pair 和 Triple 两个数据类，正如它们的名字所暗示的，Pair 数据类可包含两个任意类型的属性；Triple 可包含三个任意类型的属性。在某些希望快速开发的场景下，我们也可直接使用 Kotlin 提供的 Pair 和 Triple 两个数据类。

7.2.5 在 Lambda 表达式中解构

Kotlin 允许对 Lambda 表达式使用解构，如果 Lambda 表达式的参数是支持解构的类型（如 Pair 或 Map.Entry 等，它们都具有 operator 修饰的 componentN()方法），那么即可通过将它们放在括号中引入多个新参数来代替单个参数。例如，如下两种写法是一样的：

```
map.mapValues { entry -> "${entry.value}!" }
// 使用解构，将 entry 解构成(key,value)
map.mapValues { (key, value) -> "$value!" }
```

请注意 Lambda 表达式包含两个参数和使用解构的区别：

```
{ a -> … } // 一个参数
{ a, b -> … } // 两个参数
{ (a, b) -> … } // 一个解构对
{ (a, b), c -> … } // 一个解构对和第三个参数
```

从上面代码不难看出，Lambda 表达式的多个参数是不需要使用圆括号的，只要看到在 Lambda 表达式的形参列表中出现圆括号，那就是使用解构。

类似的，如果希望只使用后面几个 componentN()方法的返回值，则可使用下画线来代替。例如，如下代码只使用 Map 的 value。

如果解构的参数中的一个组件未使用，那么可以将其替换为下画线，以避免编造其名称。

```
map.mapValues { (_, value) -> "$value!" }
```

7.3 属性和字段

属性是Kotlin的一个重要特色，Kotlin的属性相当于Java的字段（field）再加上getter和setter方法（只读属性没有setter方法），而且开发者不需要自己实现getter和setter方法。

7.3.1 读写属性和只读属性

Kotlin使用val定义只读属性，使用var定义读写属性，系统会为只读属性生成getter方法，会为读写属性生成getter和setter方法。

在定义Kotlin的普通属性时，需要程序员显式指定初始值：要么在定义时指定初始值，要么在构造器中指定初始值。

例如，下面代码定义了一个Address类，并为该类定义了多个属性。

程序清单：codes\07\7.3\Address.kt

```
class Address {
    var street: String = ""
    var city = ""
    var province = ""
    var postCode: String? = null
}
```

上面4行粗体字代码为Address定义了4个属性。

在定义属性时，如果系统可根据属性初始值推断出属性的类型，那么程序就可以不显式指定属性的类型。

对于上面的Address类，相当于如下Java类。

```
public class Address {
    private String street: String;
    private String city;
    private String province;
    private String postCode;
    // 无参数的构造器
    public Address(){
    }
    // city的setter和getter方法
    public final void setCity(String city){
        this.city = city;
    }
    public final String getCity(){
        return this.city;
    }
    // province的setter和getter方法
    public final void setProvince(String province){
        this.province = province;
    }
    public final String getProvince(){
        return this.province;
    }
    // postCode的setter和getter方法
    public final void setPostCode(String postCode){
        this.postCode = postCode;
    }
```

```
    public final String getPostCode(){
        return this.postCode;
    }
}
```

从上面代码可以看出，Kotlin 定义一个属性，就相当于定义一个 Java 类的 private 修饰的 field，以及 public、final 修饰的 getter 和 setter 方法。

接下来程序即可使用点语法来操作属性。例如如下代码。

程序清单：codes\07\7.3\AddressTest.kt

```
fun main(args: Array<String>) {
    var addr = Address()
    // 通过点语法对属性赋值，实际就是调用 setter 方法
    addr.street = "大岗工业路"
    addr.city = "广州"
    // 通过点语法访问属性，实际就是调用 getter 方法
    println(addr.city)
    println(addr.street)
}
```

需要指出的是，虽然 Kotlin 确实会为属性生成 getter、setter 方法，但由于源程序中并未真正定义这些 getter、setter 方法，因此 Kotlin 程序不允许直接调用 Address 对象的 getter、setter 方法。但如果使用 Java 程序来调用 Address 类，由于该 Address 类中各属性对应的 field 都用了 private 修饰，因此不能用点语法直接访问这些 field，所以只能用 getter、setter 方法来访问属性。

注意：

在 Kotlin 类中定义属性后，被 Kotlin 程序使用时只能使用点语法访问属性；被 Java 程序使用时只能通过 getter、setter 方法访问属性。



使用 val 声明的是只读属性，则只有 getter 方法，没有 setter 方法。只读属性的语法和读写属性的语法有两方面的不同：①只读属性用 val 定义，读写属性用 var 定义。②只读属性不允许有 setter 方法。

例如如下 Item 类。

程序清单：codes\07\7.3\Item.kt

```
class Item(barCode: String, name: String, price: Double){
    // 定义属性，使用主构造器的参数为它们分配初始值
    val barCode = barCode
    val name = name
    val price = price
}
```

上面程序为 Item 类定义了 3 个只读属性，只读属性只有 getter 方法，没有 setter 方法，因此上面的类相当于如下 Java 类。

```
public class Item{
    private String barCode;
    private String name;
```

```
    private double price;
    // 初始化全部成员变量的构造器
    public Item(String barCode , String name , double price){
        this.barCode = barCode;
        this.name = name;
        this.price = price;
    }
    // barCode 的 getter 方法
    public final String getBarCode(){
        return this.barCode;
    }
    // name 的 getter 方法
    public final String getName(){
        return this.name;
    }
    // price 的 getter 方法
    public final double getPrice(){
        return this.price;
    }
}
```

由于上面 Item 类定义了一个主构造器，因此系统不再为该类自动提供无参数的构造器。此外，由于程序为 Item 类定义了 3 个只读属性，因此系统为它们生成 3 个 final 修饰的 getter 方法。

Kotlin 程序使用 Item 类时，同样只能用点语法来获取其属性值；但如果是 Java 程序使用 Item 类，则需要使用 getter 方法来获取属性值。如下程序示范了 Kotlin 程序使用 Item 类。

程序清单：codes\07\7.3\ItemTest.kt

```
fun main(args: Array<String>) {
    // 调用有参数的构造器创建实例
    var im = Item("1333", "疯狂 Kotlin 讲义", 69.8)
    // 通过点语法访问对象的属性
    println(im.barCode)
    println(im.name)
    println(im.price)
}
```

由于上面的 barCode、name、price 都是只读属性，因此这些属性都没有对应的 setter 方法，故程序不允许对它们赋值。

7.3.2 自定义 getter 和 setter

在定义属性时可指定自定义的 getter 和 setter 方法，这些方法可加入自己的控制逻辑，其中 getter 是一个形如 get(){}的方法（当然也可使用单表达式方法体），getter 应该是无参数、带一个返回值的方法；setter 是一个形如 set(value){}的方法（当然也可使用单表达式方法体），setter 应该是带一个参数、无返回值的方法。



注意：

定义 getter、setter 方法时无须使用 fun 关键字。



下面程序定义了一个简单的 User 类，该类包括自定义的 getter 方法。

程序清单：codes\07\7.3\User.kt

```
class User(first: String, last: String){
    var first:String = first
    var last :String = last
    val fullName: String
        // 自定义 getter 方法
        get() {
            println("执行 fullName 的 getter 方法")
            return "${first}.${last}"
        }
}
```

上面方法定义了 first 和 last 两个属性，Kotlin 会为这两个属性分别生成一个 field、getter 和 setter 方法。

程序中粗体字代码为 User 类定义了 fullName 属性，并为该属性重新定义了 getter 方法。由于该属性是一个只读属性，因此系统不需要为它生成 setter 方法。需要指出的是，由于 fullName 并不需要真正存储状态，它的返回值其实是通过 first 和 last 两个属性计算出来的，因此 Kotlin 也不需要为其生成对应的 field。当 Kotlin 不需要为该属性生成对应的 field 时，也就不能为该属性指定初始值，所以上面程序没有为 fullName 属性指定初始值。

下面程序示范了如何使用该 User 类。

程序清单：codes\07\7.3\UserTest.kt

```
fun main(args: Array<String>) {
    var user = User("悟空" , "孙")
    // 输出 user.fullName，实际上是调用其 getter 方法返回值
    println(user.fullName)
}
```

运行上面程序，可看到如下输出：

```
执行 fullName 的 getter 方法
悟空.孙
```

从上面输出可以看出，程序访问 user.fullName 时实际上就是调用了它的 fullName 的 getter 方法。

上面程序中 User 类的 fullName 是一个只读属性，其实程序也可将其定义成一个读写属性。如果 fullName 是一个读写属性，那么程序既可重写它的 getter 方法，也可重写它的 setter 方法。



提示：

对于只读属性来说，由于该属性只有 getter 方法，因此只能重写 getter 方法；对于读写属性来说，由于该属性既有 getter 方法，也有 setter 方法，因此开发者可根据需要重写其中之一，也可同时重写它们。

例如如下程序。

程序清单：codes\07\7.3\User2.kt

```
class User2(first: String, last: String){
    var first:String = first
    var last :String = last
```

```
    // 由于可通过 getter 方法推断出该属性的类型，因此可省略类型声明
    var fullName
        // 使用单表达式定义 getter 方法的方法体
        get() = "${first}.${last}"
        set(value) {
            println("执行 fullName 的 setter 方法")
            // value 字符串中不包含“.”或包含几个“.”都不行
            if ("." !in value || value.indexOf(".") != value.lastIndexOf(".")) {
                println("您输入的 fullName 不合法")
            } else {
                var tokens = value.split(".")
                first = tokens[0]
                last = tokens[1]
            }
        }
}
```

上面程序将 fullName 定义成一个读写属性，程序重写了它的 getter 方法。由于 getter 方法的方法体非常简单，故此处使用单表达式作为方法体。如果 Kotlin 可根据 getter 方法的返回值推断出该属性的类型，那么在定义该属性时也可省略声明属性的类型。

此外，该程序还为 fullName 属性定义了 setter 方法，该 setter 方法会将程序传入的字符串分解成两个部分，分别赋值给 first 和 last 属性。

从上面介绍可以看出，虽然 fullName 是一个读写属性，但其实该属性并不保存状态，因此 Kotlin 依然不需要为该属性生成 field，故该属性同样不能指定初始值。



提示：

如果读者有 Swift 编程经验，不难发现这种无须生成对应 field 的属性，其实就相当于 Swift 语言中的计算属性。这种属性并不真正存储任何状态，它的值其实是通过某种算法计算得到的，当程序对该属性赋值时，被赋的值也会被存储到其他 field 中。

下面程序示范了如何使用该 User2 类。

程序清单：codes\07\7.3\User2Test.kt

```
fun main(args: Array<String>) {
    var u = User2("悟空", "孙")
    // 使用点语法赋值，实际上是调用 setter 方法
    u.fullName = "八戒.猪"
    println(u.first)
    println(u.last)
}
```

运行该程序，可看到如下输出：

```
执行 fullName 的 setter 方法
八戒
猪
```

如果仅需要改变 getter 或 setter 方法的可见性或者对其添加注解，但不需要改变默认的实现，那么 Kotlin 允许只自定义 getter 或 setter 方法名，而不重新定义其代码实现。例如如下代码：

```
var foo: String = "abc"
```

```
    private set // 将 setter 方法改为 private 修饰，但依然使用默认实现
var bar: Int? = null
    @Inject set // 使用@Inject 修饰 setter 方法，但依然使用默认实现
```

7.3.3 幕后字段

前面已经多次提到，在 Kotlin 中定义一个普通属性时，Kotlin 会为该属性生成一个 field（字段）、getter 和 setter 方法（只读属性没有 setter 方法）。Kotlin 为该属性所生成的 field 就被称为幕后字段（backing field）。

此处有一点需要强调：如果 Kotlin 类的属性有幕后字段，则 Kotlin 要求为该属性显式指定初始值——要么在定义时指定，要么在构造器中指定；如果 Kotlin 类的属性没有幕后字段，则 Kotlin 不允许为该属性指定初始值（这是理所当然的，由于没有 field，即使指定了初始值也没地方保存）。

那么 Kotlin 何时会为属性生成幕后字段呢？只要满足以下条件，系统就会为属性生成幕后字段。

- 该属性使用 Kotlin 自动生成的 getter 和 setter 方法或其中之一。换句话说，对于只读属性，必须重写 getter 方法；对于读写属性，必须重写 getter、setter 方法；否则总会为该属性生成幕后字段。
- 重写 getter、setter 方法时，使用 field 关键字显式引用了幕后字段。

通过上面描述可以发现，Kotlin 允许开发者在 getter 或 setter 方法中通过 field 引用系统自动生成的字段（幕后字段）。例如，有时候我们希望对用户设置的属性值进行控制，此时就可以重写 setter 方法，并在 setter 方法中加入自己的控制。例如如下代码。

程序清单：codes\07\7.3\Person.kt

```
class Person(name: String, age: Int){
    // 使用 private 修饰属性，将这些属性隐藏起来
    var name = name
        set(newName){
            // 执行合理性校验，要求用户名必须在 2~6 位之间
            if (newName.length > 6 || newName.length < 2){
                println("您设置的人名不符合要求")
            } else {
                field = newName
            }
        }
    var age = age
        set(newAge){
            // 执行合理性校验，要求用户年龄必须在 0~100 之间
            if (newAge > 100 || newAge < 0) {
                println("您设置的年龄不合法")
            } else {
                field = newAge
            }
        }
}
```

上面程序定义了 name、age 两个属性。由于希望对这两个属性执行合理性校验，因此程序重写了它们的 setter 方法，上面粗体字代码就是重写 name、age 两个属性的 setter 方法的代码。

当程序重写 getter 或 setter 方法时，不能通过点语法来对 name、age 赋值！假如在 name 的 setter 方法中使用点语法对 name 赋值，由于点语法赋值的本质是调用 setter 方法，这样就会造成在 setter 方法中再次调用 setter 方法，从而形成无限递归。因此，上面程序重写 name、age 属性的 setter 方法时，实际上是通过 field 引用幕后字段，从而实现对幕后字段的赋值的。

注意：

在 getter、setter 方法中需要通过 field 关键字引用幕后字段。



下面是 Person 类的测试程序。

程序清单：codes\07\7.3\PersonTest.kt

```
fun main(args: Array<String>) {
    var p = Person("李刚", 29)
    p.age = 120 // 赋值非法，赋值失败
    println(p.age) // 输出 29
    p.age = 25 // 赋值合法，赋值成功
    println(p.age) // 输出 25
}
```

上面程序对 Person 对象的属性赋值的本质就是调用该对象的 setter 方法，因此 setter 方法的控制逻辑会发挥作用。运行上面程序，可看到如下输出：

```
您设置的年龄不合法
29
25
请按任意键继续. . .
```

从上面的运行结果可以看出，第一次对 Person 对象的 age 赋值为 120 是不合法的，因此程序输出 Person 对象的 age 依然是 29。

7.3.4 幕后属性

在个别情况下，开发者希望自己定义 field，并为该 field 提供 setter、getter 方法，就像 Java 所使用的方法。此时可使用 Kotlin 的幕后属性。

幕后属性就是用 private 修饰的属性，Kotlin 不会为幕后属性生成任何 getter、setter 方法。因此程序不能直接访问幕后属性，必须由开发者为幕后属性提供 getter、setter 方法。

例如如下程序。

程序清单：codes\07\7.3\BackingProperty.kt

```
class BackingProperty(name: String){
    // 定义 private 修饰的属性，该属性是幕后属性
    private var _name: String = name
    var name
        // 重写 getter 方法，返回幕后属性的值
        get() = _name
        set(newName){
            // 执行合理性校验，要求用户名必须在 2~6 位之间
            if (newName.length > 6 || newName.length < 2){
                println("您设置的人名不符合要求")
            } else {
                // 对幕后属性赋值
```

```
            _name = newName
        }
    }
}
fun main(args: Array<String>) {
    var p = BackingProperty("李刚")
    // 访问p.name，实际上会转为访问幕后属性_name
    println(p.name)
    // 对p.name赋值，实际上会转为对幕后属性_name赋值
    p.name = "孙悟空"
    println(p.name)
}
```

上面程序中粗体字代码定义了一个private修饰的_name属性，该属性就是一个幕后属性，Kotlin不会为该属性生成getter、setter方法，因此程序无法直接访问Person对象的_name属性。

接下来程序定义了一个name属性，并重写了name属性的getter、setter方法，重写的getter、setter方法实际上访问的是_name幕后属性。

上面这种方式就是Java的做法：先定义一个private修饰的字段，然后再为该字段定义public修饰的getter、setter方法。这种方式比较烦琐，通常没有必要采用如此烦琐的方式。

7.3.5 延迟初始化属性

正如前面所介绍的，Kotlin要求所有属性必须由程序员显式初始化——要么在定义该属性时赋初始值；要么在构造器中对该属性赋初始值。但在某些时候，这不是必需的。比如，我们可能通过依赖注入为属性设置初始值，或者在单元测试的setUp方法中初始化该属性……总之，并不需要在定义属性时或在构造器中对属性执行初始化。

Kotlin提供了lateinit修饰符来解决属性的延迟初始化。使用lateinit修饰的属性，可以在定义该属性时和在构造器中都不指定初始值。

对lateinit修饰符有以下限制。

- lateinit只能修饰在类体中声明的可变属性（使用val声明的属性不行，在主构造器中声明的属性也不行）。
- lateinit修饰的属性不能有自定义的getter或setter方法。
- lateinit修饰的属性必须是非空类型。
- lateinit修饰的属性不能是原生类型（即Java的8种基本类型对应的类型）。

与Java不同的是，Kotlin不会为属性执行默认初始化。因此，如果在lateinit属性赋初始值之前访问它，程序将会引发“lateinit property name has not been initialized”异常。

例如，如下程序定义了两个延迟初始化的属性。

程序清单：codes\07\7.3\LateinitTest.kt

```
import java.util.Date
class User{
    // 延迟初始化属性
    lateinit var name: String
    lateinit var birth: Date
}
fun main(args: Array<String>){
    var user = User()
//    println(user.name)
```

```
//    println(user.birth)
    user.name = "孙悟空"
    user.birth = Date()
    println(user.name)
    println(user.birth)
}
```

上面程序中两行粗体字代码定义了 name、birth 两个 lateinit 属性，这样即可在定义该属性时和在构造器中都不为其指定初始值。

由于 User 的 name、birth 两个属性都没有初始值，因此程序创建 User 对象后不能立即访问它的 name、birth 属性，否则将会引发异常，如上面程序中两行注释掉的代码所示。

7.3.6 内联属性

从 Kotlin 1.1 开始，inline 修饰符可修饰没有幕后字段的属性的 getter 或 setter 方法，既可单独修饰属性的 getter 或 setter 方法；也可修饰属性本身，这相当于同时修饰该属性的 getter 和 setter 方法。

对于使用 inline 修饰的 getter、setter 方法，就像前面介绍的内联函数一样，程序在调用 getter 或 setter 方法时也会执行内联化。

如下程序示范了内联属性。

程序清单：codes\07\7.3\InlinePropTest.kt

```
class Name(name: String, desc: String){
    var name = name;
    var desc = desc;
}
class Product{
    var productor: String? = null
    // inline 修饰属性的 getter 方法，表明读取属性时会内联化
    val proName: Name
        inline get() = Name("疯狂 Kotlin 讲义", "最系统的 Kotlin 书")
    // inline 修饰属性的 setter 方法，表明设置属性时会内联化
    var author: Name
        get() = Name("李刚", "无")
        inline set(v){
            this.productor = v.name
        }
    // inline 修饰属性本身，表明读取和设置属性时都会内联化
    inline var pubHouse: Name
        get() = Name("电子工业出版社", "无")
        set(v){
            this.productor = v.name
        }
}
```

上面程序为 Product 定义了 3 个属性：proName、author 和 pubHouse，这 3 个属性都是没有幕后字段的属性，因此 Kotlin 允许将它们变成内联属性。程序中第一行粗体字代码使用 inline 修饰了 proName 属性的 getter 方法，这意味着当程序读取该属性的值时会变成内联化；程序中第二行粗体字代码使用 inline 修饰了 author 属性的 setter 方法，这意味着当程序设置该属性时会变成内联化；程序中第三行粗体字代码使用 inline 修饰了 pubHouse 属性本身，这意味着当程序读取和设置该属性时都会变成内联化。

7.4 隐藏和封装

封装（Encapsulation）是面向对象的三大特征之一（另外两个特征是继承和多态），指的是将对象的状态信息隐藏在对象内部，不允许外部程序直接访问对象内部信息，而是通过该类所提供的方法来实现对内部信息的操作和访问。

封装是面向对象编程语言对客观世界的模拟，在客观世界里，对象的状态信息都被隐藏在对象内部，外界无法直接操作和修改。对一个类或对象实现良好的封装，可以实现以下目的。

- 隐藏类的实现细节。
- 让使用者只能通过事先预定的方法来访问数据，从而可以在该方法里加入控制逻辑，限制对属性的不合理访问。
- 可进行数据检查，从而有利于保证对象信息的完整性。
- 便于修改，提高代码的可维护性。

为了实现良好的封装，需要从两个方面考虑。

- 将对象的属性和实现细节隐藏起来，不允许外部直接访问。
- 把方法暴露出来，让方法来控制对这些属性进行安全的访问和操作。

因此，封装实际上有两个方面的含义：把该隐藏的隐藏起来，把该暴露的暴露出来。封装需要通过使用包和访问权限修饰符（也称为可见性修饰符，Visibility Modifier）来实现。

7.4.1 包和导包

Kotlin 的包与 Java 的包相同，既是逻辑上的一个程序单元，也是一个命名空间。如果希望把函数、类放在指定的包结构下，则应该在 Kotlin 源程序的第一个非注释行放置如下格式的代码：

```
package packagename
```

一旦在 Kotlin 源程序中使用了这条 package 语句，就意味着该源程序中定义的所有类、函数都属于这个包。位于包中的每个类的完整类名都应该是包名和类名的组合，如果其他人需要使用该包中的类，也应该使用包名加类名的组合。

下面程序在 lee 包中定义了一个类和一个函数。

程序清单：codes\07\7.4\Hello.kt

```
package lee
fun test(){
    println("简单的test()函数")
}
class Foo(name: String){
    var name = name
}
fun main(args: Array<String>){
    println("带包的主函数")
}
```

上面程序中粗体字代码表明把 test()函数和 Foo 类都放在 lee 包空间下。把上面的源文件保存在任意位置，编译这个源文件会多出一个名为 lee 的文件夹，该文件夹中则有 Foo.class 和 HelloKt.class 文件。

这一点和 Java 的设计相同：位于包中的类，在文件系统中也必须有与包名层次相同的目录结构。因此上面的 Foo 类、HelloKt 类必须放在 lee 文件夹中才是有效的。



提示：

kotlinc 编译器更方便，它甚至不像 javac 编译器那样需要-d 选项才会为包生成文件目录，kotlinc 编译器总会为源文件中的包结构生成对应的文件目录。

一旦为 Kotlin 源程序指定包名之后，该源程序中的函数和类就都具有了指定的包名。因此，上面程序中 test()函数的完整函数名是 lee.test()；Foo 类的完整类名是 lee.Foo。

前面已经讲过，运行 Kotlin 程序实际上就是运行 kotlinc 为主函数所生成的主类，该主类是 HelloKt，由于它的包名是 lee，因此其完整类名应该是 lee.HelloKt。进入编译器生成的 lee 文件夹所在的路径下，执行如下命令：

```
java lee.HelloKt
```

可以看到上面程序正常输出。

需要说明的是，如果希望使用 Java 来调用上面的 test()函数，由于 Kotlin 实际上还会为上面程序生成一个 HelloKt 类，而 test()函数将以 HelloKt 类的类方法而存在，因此程序应该通过 lee.HelloKt.test()的形式来调用 test()函数。

与 Java 的包机制完全相同的是，Kotlin 的包也需要两个方面的保证：①在源程序中使用 package 语句指定包名；②必须将 class 文件放在对应的路径下。

从语句的角度来看，只要求包名是有效的标识符即可；但从可读性规范的角度来看，包名应该全部是小写字母，而且应该由一个或多个有意义的单词连缀而成。

通常来说，源程序的包名大致按如下规则命名：

```
公司域名倒写.项目名.模块名.组件类型
```

为了使用其他包中的函数和类，Kotlin 同样使用 import 执行导入。

Kotlin 的 import 语法和 Java 类似，同样支持精确导入和通配符导入。

精确导入：

```
import foo.Bar // 导入 foo 包中的 Bar，以后使用 Bar 时无需包名
```

通配符导入：

```
import foo.* // 导入 foo 包中的所有成员，以后使用 foo 包中的所有成员时都无需包名
```

Kotlin 的 import 功能比较强大，它相当于 Java 的 import 和 import static（静态导入）的合体，它不仅可以导入类，还可以导入如下内容。

- 顶层函数及属性。
- 在对象声明中声明的函数和属性。
- 枚举常量。

此外，如果需要在同一个源文件中导入不同包中的同名类（比如 java.util.Date 和 java.sql.Date），以前 Java 程序很难处理这种情况，通常有一个类总要使用全限定类名（放弃导包），但 Kotlin 提供了更简单的处理方式。看如下代码。

程序清单：codes\07\7.4\DiffImport.kt

```
import java.util.Date
// 导入 java.sql.Date，并指定别名为 SDate
```

```
import java.sql.Date as SDate
fun main(args: Array<String>) {
    // 使用 java.util.Date
    var d = Date()
    // 使用 java.sql.Date
    var d2 = SDate(System.currentTimeMillis())
    println(d)
    println(d2)
}
```

从上面粗体字代码可以看出，Kotlin 的 import 语句支持 as 关键字，这样就可以为导入类指定别名，从而解决了导入两个不同包中的同名类的问题。

7.4.2 Kotlin 的默认导入

正如前面所看到的，前面程序使用 Kotlin 定义的 List、Map 等类时，程序都没有为这些类导包，其原因都是因为 Kotlin 的默认导入。

Kotlin 默认会导入如下包。

- kotlin.*
- kotlin.annotation.*
- kotlin.collections.*
- kotlin.comparisons.*（自 Kotlin 1.1 起）
- kotlin.io.*
- kotlin.ranges.*
- kotlin.sequences.*
- kotlin.text.*

此外，对于 JVM 平台，还会自动导入如下两个包。

- java.lang.*
- kotlin.jvm.*

对于 JavaScript 平台，则额外导入如下包。

- kotlin.js.*

7.4.3 使用访问控制符

Kotlin 提供了 4 个访问控制符：private、internal、protected 和 public，分别代表 4 个访问控制级别。Kotlin 的 4 个访问控制符的意义如下。

- **private**：与 Java 的 private 类似，private 成员只能在该类的内部或文件的内部被访问。
- **internal**：internal 成员可以在该类的内部或文件的内部或者同一个模块内被访问。
- **protected**：protected 成员可以在该类的内部或文件的内部或者其子类中被访问。
- **public**：public 成员可以在任意地方被访问。

与 Java 不同的是，如果 Kotlin 没有显式指定修饰符的话，默认的访问控制修饰符是 public。

根据上面描述不难发现，Kotlin 的访问控制符与 Java 的区别有如下几点。

- Kotlin 取消了 Java 的默认访问权限（包访问权限），引入了 internal 访问控制符（模块访问权限）。
- Kotlin 取消了 protected 的包访问权限。

- Kotlin 的默认访问控制符是 public。

此处涉及一个问题：何为模块？模块是编译在一起的一套 Kotlin 文件，模块的存在形式有如下几种。

- 一个 IntelliJ IDEA 模块。
- 一个 Maven 项目。
- 一个 Gradl 源集。
- 一次<kotlinc>的 Ant 任务执行所编译的一套文件。



提示：

Kotlin 使用 internal 代替 Java 的默认访问控制符是很好的改进。Java 的包访问控制符通常很不方便使用，例如同一个项目组开发一个模块，同一个模块内不同类型的应用组件肯定放在不同的子包中（比如数据库访问组件放在 dao 子包中，业务逻辑组件放在 service 子包中），此时如果同一个模块中的不同成员使用的是包访问控制符，那么它们不能互相访问，故只能改为使用 public 修饰符，但使用 public 访问控制符又彻底暴露了这些成员，所以 Java 缺一个模块访问权限，而包访问权限实在是一个“鸡肋”。因此，Kotlin 使用 internal 代替包访问权限确实是一个不错的改进。

下面详述 Kotlin 不同作用域中的成员可支持的访问控制符。

1. 位于包内的顶层成员

对于位于包内的顶层成员（包括顶层类、接口、函数、属性），只能使用 private、internal 和 public 其中之一，不能使用 protected 修饰符。

- 使用 private 修饰，这些顶层成员只能在当前文件中被访问。
- 使用 internal 修饰，这些顶层成员只能在当前文件或当前模块中被访问。
- 不加修饰符或使用 public 修饰，这些顶层成员可以在任意地方被访问。

例如，如下程序示范了包内的顶层成员的访问权限。

```
// 文件名：example.kt
package lee

private fun foo() {} // 该函数仅在 example.kt 内可访问
internal fun test() {} // 该函数可在相同模块内被访问
fun info() {} // 该函数可在任意地方被访问

public var bar: Int = 5 // 该属性可在任意地方被访问
    private set // setter 仅在 example.kt 内可访问

internal val baz = 6 // 该属性可在相同模块内被访问
    private set  // setter 仅在 example.kt 内可访问
```

需要说明的是，属性的 getter 方法的访问权限总是与该属性保持一致。



提示：

直接定义在包内的顶层函数、属性，会转换成 Kotlin 所生成的类（类名为文件名+Kt 后缀的类）中的静态方法和静态属性。

2. 位于类、接口之内的成员

对于位于类、接口之内的成员（包括顶层类、接口、函数、属性），能使用 private、internal、protected 和 public 其中之一。

- 使用 private 修饰，这些成员只能在该类中被访问。
- 使用 internal 修饰，这些成员能在该类或当前模块中被访问。
- 使用 protected 修饰，这些成员能在该类或该类的子类中被访问。
- 不加修饰符或使用 public 修饰，这些成员可以在任意地方被访问。

如果重写一个 protected 成员，且没有显式指定访问权限修饰符，那么该成员依然是 protected 访问权限。

程序清单：codes\07\7.4\ModifierTest.kt

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal val c = 3
    val d = 4  // 默认是 public 访问权限

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a 不可访问
    // b、c、d 可访问
    // Nested 和 e 可访问

    override val b = 5   // 被重写的 b 依然是 protected 访问权限
}

class Other(o: Outer) {
    // o.a、o.b 不可访问
    // o.c 与 Other 类在同一个模块中，可以被访问
    // o.d 可访问
    // Outer.Nested 不可访问，Nested::e 也不可访问
}
```

类的主构造器有点特别，它是在类头部分声明的。如果需要为主构造器指定访问权限修饰符，则一定要使用 constructor 关键字，并在该关键字前面添加 private、internal、protected 和 public 其中之一。

如果不为主构造器指定访问权限修饰符，那么主构造器默认的访问权限修饰符是 public。

与 Java 相同的是，局部声明（包括局部变量、局部方法、局部嵌套类）的作用域仅在该方法（或函数）内有效，使用访问控制符没有任何意义，因此不能使用 private、internal、protected、public 修饰符。

7.5 深入构造器

构造器用于在创建实例时执行初始化。构造器是创建对象的重要途径（即使使用工厂模式、

反射等方式创建对象，其实质也依然是依赖于构造器），因此 Kotlin 类必须包含一个或一个以上的构造器。

7.5.1 主构造器和初始化块

前面已经提到，Kotlin 类可以定义 0~1 个主构造器和 0~*N* 个次构造器。如果主构造器没有任何注解或可见性修饰符，则可以省略 constructor 关键字。

主构造器作为类头的一部分，可以声明形参，但它自己并没有执行体。那么主构造器的形参有什么用呢？其作用主要有两点。

- 初始化块可以使用主构造器定义的形参。
- 在声明属性时可以使用主构造器定义的形参。

由此可见，Kotlin 的主构造器并不是传统意义上的构造器，它更像 Java 的初始化块，或者说是对初始化块的增强——Java 的初始化块不能传入参数；Kotlin 通过主构造器的设计，允许为初始化块传入参数。

初始化块的语法格式如下：

```
init {
    // 初始化块中的可执行代码，可以使用主构造器定义的参数
    ...
}
```

初始化块中的代码可以使用主构造器定义的参数，也可以包含任何可执行语句，包括定义局部变量、调用其他对象的方法，以及使用分支、循环语句等。

下面程序定义了一个 Person 类，它既包含了主构造器，也包含了初始化块。下面看看在程序中创建 Person 对象时发生了什么。

程序清单：code\07\7.5\Person.kt

```
class Person(name: String){
    // 下面定义一个初始化块
    init{
        var a = 6
        if (a > 4){
            println("Person 初始化块：局部变量 a 的值大于 4")
        }
        println("Person 的初始化块")
        println("name 参数为：${name}")
    }
    // 定义第二个初始化块
    init{
        println("Person 的第二个初始化块")
    }
}
fun main(args: Array<String>){
    Person("孙悟空")
}
```

上面程序的 main()函数只创建了一个 Person 对象，程序输出如下：

```
Person 初始化块：局部变量 a 的值大于 4
Person 的初始化块
name 参数为：孙悟空
Person 的第二个初始化块
```

从运行结果可以看出，当程序通过主构造器创建对象时，系统其实就是调用该类里定义的初始化块，如果一个类里定义了两个普通初始化块，则前面定义的初始化块先执行，后面定义的初始化块后执行。



提示：

和 Java 类似的是，Kotlin 也允许一个类里定义两个初始化块，但这没有任何意义。因为初始化块是在创建对象时隐式执行的，而且它们总是全部执行，因此完全可以把多个普通初始化块合并成一个初始化块，从而让程序更加简洁，可读性更强。

从上面结果不难看出，程序调用主构造器创建对象，实际上就是执行初始化块。由此可见，主构造器的主要作用就是为初始化块定义参数，因此主构造器更像是初始化块的一部分。也可以说，初始化块就是主构造器的执行体。

构造器最大的用处就是在创建对象时执行初始化，但由于初始化块就是主构造器的执行体，因此，如果希望为对象的属性显式指定初始值，则也可以通过初始化块来指定。

如果程序员没有为 Kotlin 类提供任何构造器（主构造器和次构造器），则系统会为这个类提供一个无参数的主构造器，这个构造器的执行体为空，不做任何事情。无论如何，Kotlin 类至少包含一个构造器。

下面程序示范了通过初始化块对类进行自定义的初始化操作。

程序清单：codes\07\7.5\ConstructorTest.kt

```
// 提供主构造器，该构造器包含两个参数
class ConstructorTest(name: String, count: Int){
    var name: String
    var count: Int
    // 定义初始化块，它相当于主构造器的执行体
    init {
        // 初始化块中的 this 代表其进行初始化的对象
        // 下面两行代码将传入的两个参数赋给 this 所代表的对象的 name 和 count 属性
        this.name = name
        this.count = count
    }
}
fun main(args: Array<String>) {
    // 使用自定义的构造器来创建对象
    // 系统将会对该对象执行自定义的初始化
    var tc = ConstructorTest("疯狂 Java 讲义" , 90000)
    // 输出 ConstructorTest 对象的 name 和 count 两个属性
    println(tc.name)
    println(tc.count)
}
```

运行上面程序，将看到输出 ConstructorTest 对象时，它的 name 和 count 属性都是调用主构造器时传入的参数。

一旦程序员提供了自定义的构造器，系统就不再提供默认的构造器，因此上面的 ConstructorTest 类不能再通过 ConstructorTest()代码来创建实例，因为该类不再包含无参数的构造器。

如果希望该类保留无参数的构造器，或者希望有多个初始化过程，则可以为该类提供多个

构造器。如果一个类里提供了多个构造器，那么就形成了构造器重载。

7.5.2 次构造器和构造器重载

Kotlin 允许使用 constructor 关键字定义 *N* 个次构造器，次构造器类似于 Java 传统的构造器。如果读者认真阅读过《疯狂 Java 讲义》或对 Java 有深刻了解，一定会记得：Java 的初始化块其实是假象，其会自动插入到每个构造器的前面执行。

而 Kotlin 的主构造器其实属于初始化块（或者说，初始化块其实是主构造器的执行体），因此 Kotlin 要求所有的次构造器必须委托调用主构造器，可以直接委托或通过别的次构造器间接委托。所谓“委托”，其实就是要先调用主构造器（执行初始化块中的代码），然后才执行次构造器代码。



提示：

这种设计和 Java 的设计保持一致——Java 会让每个构造器（相当于 Kotlin 的次构造器）自动调用初始化块；Kotlin 则要求每个次构造器都委托调用主构造器。

由此可见，Kotlin 的初始化块和 Java 的初始化块一样，都是多个次构造器中相同的代码。图 7.1 显示了把两个构造器中的代码提取成初始化块示意图。

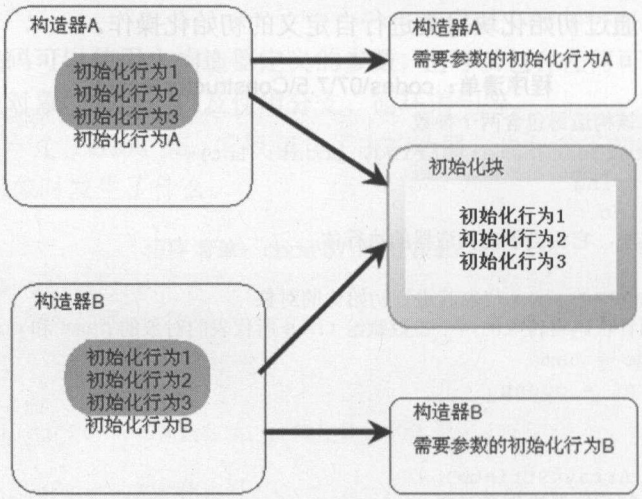


图 7.1 将构造器代码提取成初始化块示意图

从图 7.1 中可以看出，如果两个构造器中有相同的初始化代码，就可以把它们放在初始化块中定义。如果这些初始化代码需要参数，则可将参数放在主构造器中定义。通过把多个构造器中的相同代码提取到初始化块中定义，能更好地提高初始化代码的复用性，提高整个应用的可维护性。

同一个类里具有多个构造器，多个构造器的形参列表不同，即被称为构造器重载。程序可通过不同的构造器来创建对象，但不管使用哪个构造器，首先都要先调用主构造器（执行初始化块代码）。

为了让系统能区分不同的构造器，这些构造器的参数列表必须不同。

下面程序示范了构造器重载，利用构造器重载就可以通过不同的构造器来创建对象。

程序清单：codes\07\7.5\ConstructorOverload.kt

```
class ConstructorOverload{
    var name: String?
    var count: Int
    init{
        println("初始化块！")
    }
    // 提供无参数的构造器
    constructor(){
        name = null
        count = 0
    }
    constructor(name:String, count:Int){
        this.name = name
        this.count = count
    }
}
fun main(args: Array<String>) {
    // 通过无参数的构造器创建 ConstructorOverload 对象
    var oc1 = ConstructorOverload()
    // 通过有参数的构造器创建 ConstructorOverload 对象
    var oc2 = ConstructorOverload(
        "轻量级 Java EE 企业应用实战", 300000)
    println("${oc1.name} ${oc1.count}")
    println("${oc2.name} ${oc2.count}")
}
```

上面的 ConstructorOverload 类提供了两个重载的次构造器，但它们的形参列表不同。在调用构造器时，系统将根据传入的实参列表来决定调用哪个构造器。

运行上面程序，可看到如下输出：

```
初始化块！
初始化块！
null 0
轻量级 Java EE 企业应用实战 300000
```

从上面的运行结果可以看出，不管调用哪个构造器创建对象，系统总会先执行初始化块——也就是说，初始化块总会在所有次构造器之前执行。用 Kotlin 的专业术语来说，叫作：所有的次构造器都要委托调用初始化块。

从上面的运行结果可以看出，使用不同的构造器初始化出来的对象具有不同的状态。

上面的 ConstructorOverload 没有定义主构造器，因此次构造器不需要委托主构造器。下面再定义一个带主构造器的类。

程序清单：codes\07\7.5\User.kt

```
// 定义主构造器
class User(name: String){
    var name: String
    var age: Int
    var info: String? = null
    init {
        println("User 的初始化块")
        this.name = name
        this.age = 0
```

```
    }
    // 委托给主构造器
    constructor(name: String, age: Int): this(name){  // ①
        this.age = age
    }
    // 委托给(String,Int)构造器
    constructor(name: String, age: Int, info: String): this(name, age){  // ②
        this.info = info
    }
}
fun main(args: Array<String>){
    // 调用主构造器
    var us1 = User("孙悟空")
    println("${us1.name} => ${us1.age} => ${us1.info}")
    // 调用(String, Int)构造器
    var us2 = User("白骨精", 21)
    println("${us2.name} => ${us2.age} => ${us2.info}")
    // 调用(String, Int, String)构造器
    var us3 = User("蜘蛛精", 20, "吐丝织网")
    println("${us3.name} => ${us3.age} => ${us3.info}")
}
```

上面程序中 User 类定义了一个带 String 参数的主构造器，接下来程序中两行粗体字代码使用 constructor 定义了两个次构造器，其中①号粗体字代码定义的次构造器委托了主构造器；②号粗体字代码定义的次构造器委托了①号粗体字代码定义的次构造器，这样就间接委托了主构造器。

从上面代码可以看出，Kotlin 使用“:this(参数)”的语法委托另一个构造器，到底委托哪个构造器则取决于传入的参数。系统会根据传入的参数来推断委托了哪个构造器。

主函数分别使用了 3 个构造器来创建实例，运行程序可看到如下输出：

```
User 的初始化块
孙悟空 => 0 => null
User 的初始化块
白骨精 => 21 => null
User 的初始化块
蜘蛛精 => 20 => 吐丝织网
```

从上面输出可以看到，当程序调用主构造器创建实例时，主构造器自动执行初始化块；当程序调用两个次构造器创建实例时，也会先执行初始化块，这是由于它们都委托了主构造器的缘故。



提示：

熟悉《疯狂 Java 讲义》的读者一定还记得：Java 构造器使用 this(参数)来调用重载的构造器，且 this(参数)调用必须作为构造器的第一条语句。使用 this 调用重载的构造器时，系统会根据 this 后括号里的实参来推断调用哪个构造器。由此可见，Kotlin 的构造器委托其实就是 Java 构造器中的 this(参数)调用，只不过 Kotlin 要求将 this(参数)调用放在构造器声明的冒号之后，而 Java 要求将 this(参数)调用放在构造器的第一行——它们的本质是一样的。Kotlin 为之换个“新马甲”：“委托”。

7.5.3 主构造器声明属性

Kotlin 允许在主构造器上声明属性，直接在参数之前使用 var 或 val 即可声明属性——使用 var 声明的是读写属性，使用 val 声明的是只读属性。当程序调用这种方式声明的主构造器创建对象时，传给该构造器的参数将会赋值给对象的属性。

例如如下程序。

程序清单：codes\07\7.5\Item.kt

```
// 使用主构造器声明属性
class Item(val code: String, var price: Double){
}
fun main(args: Array<String>) {
    var im = Item("1234567", 6.7)
    println(im.code) // 输出"1234567"
    println(im.price) // 输出 6.7
}
```

如果主构造器的所有参数都有默认值，程序能以构造参数的默认值来调用该构造器（即不需要为构造参数传入值），此时看上去就像调用无参数的构造器。例如如下程序。

程序清单：codes\07\7.5\Customer.kt

```
// 使用主构造器声明属性
class Customer(val name: String = "匿名",
    var addr: String = "天河"){
}
fun main(args: Array<String>) {
    // 调用有参数的主构造器
    var ct = Customer("孙悟空", "花果山")
    println(ct.name) // 输出"孙悟空"
    println(ct.addr) // 输出"花果山"
    // 以构造参数的默认值调用构造器，看上去像调用无参数的构造器
    var ctm = Customer()
    println(ctm.name) // 输出"匿名"
    println(ctm.addr) // 输出"天河"
}
```

上面程序定义 Customer 类的主构造器时为所有参数都指定了默认值，因此程序能以构造参数的默认值来调用构造器（无须传入参数值），看上去就像调用无参数的构造器。

7.6 类的继承

继承是面向对象的三大特征之一，也是实现软件复用的重要手段。Kotlin 的继承同样是单继承：每个子类最多只有一个直接父类。

7.6.1 继承的语法

Kotlin 的子类继承父类的语法格式如下：

```
修饰符 class SubClass : SuperClass {
    // 类定义部分
}
```

从上面的语法格式来看，定义子类的语法非常简单，只需在原来的类定义中添加

SuperClass，即可表明该子类继承了 SuperClass 类。

从子类的角度来看，子类扩展（extend）了父类；但从父类的角度来看，父类派生（derive）出子类。也就是说，扩展和派生所描述的是同一个动作，只是观察的角度不同而已。

如果在定义一个 Kotlin 类时并未显式指定这个类的直接父类，则这个类默认扩展 Any 类。因此，Any 类是所有类的父类，要么是其直接父类，要么是其间接父类。

需要说明的是，Any 类不是 java.lang.Object 类，Any 类只有 equals()、hashCode()和 toString()这 3 个方法。

还有一点需要说明的是，Kotlin 的类默认就有 final 修饰，因此 Kotlin 的类默认是不能派生子类的。为了让一个类能派生子类，需要使用 open 修饰该类。



提示：

open 就是 final 的反义词，用于取消 Kotlin 自动添加的 final 修饰符。

下面我们尝试定义如下父子类。

```
open class BaseClass{
}
class SubClass: BaseClass{
}
```

如果尝试编译上面两个类，就会发现程序无法通过编译，这是为什么呢？

回忆《疯狂 Java 讲义》中关于子类构造器和父类构造器的介绍：子类构造器总要调用父类构造器一次。子类构造器调用父类构造器分如下几种情况。

- 子类构造器执行体的第一行使用 super(参数)显式调用父类构造器，系统将根据 super(参数)调用中传入的实参列表调用父类对应的构造器。
- 子类构造器执行体的第一行代码使用 this(参数)显式调用本类中重载的构造器，系统将根据 this(参数)调用中传入的实参列表调用本类中的另一个构造器。调用本类中的另一个构造器最终还是要调用父类构造器。
- 子类构造器执行体中既没有 super(参数)调用，也没有 this(参数)调用，系统将会在执行子类构造器之前，隐式调用父类无参数的构造器。

Kotlin 的构造器同样要遵守这个规则，只不过 Kotlin 为之换了个新说法：委托父类构造器。而且由于 Kotlin 的构造器分为主构造器和次构造器，因此情况略微复杂一些。下面分主构造器和次构造器进行详细说明。

1. 子类的主构造器

如果子类定义了主构造器，由于主构造器属于类头部分（如果不定义初始化块，它就没有执行体），为了让主构造器能调用父类构造器，因此主构造器必须在继承父类的同时委托调用父类构造器。例如如下代码。

程序清单：codes\07\7.6\SubClass.kt

```
open class BaseClass{
    var name: String
    constructor(name: String){
        this.name = name
    }
```

```
}
// 子类没有显式声明主构造器
// 子类默认有一个主构造器，因此要在声明继承时委托调用父类构造器
class SubClass1: BaseClass("foo"){
}
// 子类显式声明主构造器
// 主构造器必须在声明继承时委托调用父类构造器
class SubClass2(name: String): BaseClass(name){
}
```

上面程序为 BaseClass 派生了两个子类：SubClass1 和 SubClass2，其中 SubClass1 没有显式声明主构造器，系统会为该类自动生成一个无参数的主构造器，因此程序在继承 BaseClass 时必须立即调用父类构造器，如上面第一处粗体字代码所示。

程序为 SubClass2 显式定义了一个带参数的主构造器，因此程序同样需要在继承 BaseClass 时必须立即调用父类构造器，如上面第二处粗体字代码所示。

2. 子类的次构造器

次构造器同样需要委托调用父类构造器。

如果子类定义了主构造器，由于子类的次构造器总会委托调用子类的主构造器（直接或间接），而主构造器一定会委托调用父类构造器，因此子类的所有次构造器最终也调用了父类构造器。

如果子类没有定义主构造器，则此时次构造器委托调用父类构造器可分为 3 种方式。

- 子类构造器显式使用:this(参数)显式调用本类中重载的构造器，系统将根据 this(参数)调用中传入的实参列表调用本类中的另一个构造器。调用本类中的另一个构造器最终还是要调用父类构造器。
- 子类构造器显式使用:super(参数)委托调用父类构造器，系统将根据 super(参数)调用中传入的实参列表调用父类对应的构造器。
- 子类构造器既没有:super(参数)调用，也没有:this(参数)调用，系统将会在执行子类构造器之前，隐式调用父类无参数的构造器。

这段描述和上面 Java 子类构造器调用父类构造器的描述相似吗？相似就对了。正如前面所介绍的，Kotlin 的次构造器相当于 Java 的构造器，因此 Kotlin 的次构造器委托调用父类构造器的 3 种方式，正好对应于 Java 构造器调用父类构造器的 3 种方式。它们唯一的区别只是写法不同：Java 将调用父类构造器写在方法体内，Kotlin 将委托父类构造器写在构造器声明中。

如下代码示范了没有主构造器的子类的次构造器是如何调用父类构造器的。

程序清单：codes\07\7.6\Sub.kt

```
open class Base{
    constructor(){
        println("Base 的无参数的构造器")
    }
    constructor(name: String){
        println("Base 的带一个 String 参数:${name}的构造器: ")
    }
}
class Sub: Base{
    // 构造器没有显式委托
    // 因此该次构造器将会隐式委托调用父类无参数的构造器
```

```
    constructor(){
        println("Sub 的无参数的构造器")
    }
    // 构造器用 super(name)显式委托父类带 String 参数的构造器
    constructor(name: String): super(name){
        println("Sub 的 String 构造器, String 参数为: ${name}")
    }
    // 构造器用 this(name)显式委托本类中带 String 参数的构造器
    constructor(name:String, age: Int): this(name){
        println("Sub 的 String,Int 构造器, Int 参数为: ${age}")
    }
}
fun main(args: Array<String>){
    Sub()
    Sub("Sub")
    Sub("子类", 29)
}
```

上面的 Sub 类没有定义主构造器，类体中 3 行粗体字代码定义了 3 个次构造器。第一个次构造器既没有 super(参数)委托，也没有(this)委托，因此该构造器会隐式委托调用父类无参数的构造器。

第二个次构造器使用:super(name)委托调用，其中 name 是一个 String 类型的参数，因此该构造器属于显式委托调用父类带一个 String 参数的构造器。

第三个次构造器使用:this(name)委托调用，其中 name 是一个 String 类型的参数，因此该构造器属于显式委托调用该类中带一个 String 参数的构造器，即调用前一个构造器；而前一个构造器委托调用了父类构造器，因此该次构造器最终也调用了父类构造器一次。

注意：

Kotlin 与 Java 的设计相同：*所有子类构造器必须调用父类构造器一次。*



当调用子类构造器来初始化子类对象时，父类构造器总会在子类构造器之前执行；不仅如此，在执行父类构造器时，系统会再次上溯执行其父类构造器……依此类推，创建任何 Kotlin 对象，最先执行的总是 Any 类的构造器。

7.6.2 重写父类的方法

子类继承父类，将可以获得父类的全部属性和方法。

下面程序示范了子类继承父类的特点。如下是 Fruit 类的代码。

程序清单：codes\07\7.6\Fruit.kt

```
open class Fruit(var weight: Double){
    fun info(){
        println("我是一个水果! 重${weight}g! ")
    }
}
```

接下来定义该 Fruit 类的子类 Apple，程序如下。

程序清单：codes\07\7.6\Apple.kt

```
// 定义 Apple 类继承 Fruit
```

```
class Apple: Fruit(0.0)
fun main(args: Array<String>) {
    // 创建 Apple 对象
    var a = Apple()
    // Apple 对象本身没有 weight 属性
    // 因为 Apple 的父类有 weight 属性，所以也可以访问 Apple 对象的 weight 属性
    a.weight = 56.0
    // 调用 Apple 对象的 info()方法
    a.info()
}
```

上面的 Apple 类只是一个空类，但程序中创建了 Apple 对象之后，可以访问该 Apple 对象的 weight 属性和 info()方法，这表明 Apple 对象也具有了 weight 属性和 info()方法，这就是继承的作用。

子类继承了父类，子类是一个特殊的父类。大部分时候，子类总是以父类为基础，额外增加新的属性和方法。但有一种情况例外：子类需要重写父类的方法。例如，鸟类都包含了飞翔方法，其中鸵鸟是一种特殊的鸟类，因此鸵鸟应该是鸟的子类，它也将从鸟类获得飞翔方法，但这个飞翔方法明显不适合鸵鸟，为此，鸵鸟需要重写鸟类的方法。

下面程序先定义了一个 Bird 类。

程序清单：codes\07\7.6\Bird.kt

```
open class Bird{
    // Bird 类的 fly()方法
    open fun fly(){
        println("我在天空里自由自在地飞翔...")
    }
}
```

留意上面的 fly()方法，该方法同样使用了 open 修饰符，这依然是由 Kotlin 的设计哲学所决定的——Kotlin 默认为所有方法添加 final 修饰符，阻止该方法被重写，添加 open 关键字用于阻止 Kotlin 自动添加 final 修饰符。

下面再定义一个 Ostrich 类，这个类扩展了 Bird 类，重写了 Bird 类的 fly()方法。

程序清单：codes\07\7.6\Ostrich.kt

```
class Ostrich: Bird(){
    // 重写 Bird 类的 fly()方法
    override fun fly(){
        println("我只能在地上奔跑...")
    }
}
fun main(args: Array<String>) {
    // 创建 Ostrich 对象
    var os = Ostrich()
    // 执行 Ostrich 对象的 fly()方法，将输出"我只能在地上奔跑..."
    os.fly()
}
```

运行上面程序，将看到运行 os.fly()时执行的不再是 Bird 类的 fly()方法，而是执行 Ostrich 类的 fly()方法。

从上面的粗体字代码可以看出，Kotlin 类重写父类的方法必须添加 override 修饰符，就像 Java 的@Override 注解，只不过 Java 的@Override 是可选的，而 Kotlin 的 override 修饰符是强

制的。

这种子类包含与父类同名方法的现象被称为方法重写（Override），也被称为方法覆盖。可以说子类重写了父类的方法，也可以说子类覆盖了父类的方法。

方法的重写要遵循“两同两小一大”规则，“两同”即方法名相同、形参列表相同；“两小”指的是子类方法的返回值类型应比父类方法的返回值类型更小或相等，子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相等；“一大”指的是子类方法的访问权限应比父类方法的访问权限更大或相等。

当子类覆盖了父类方法后，子类的对象将无法访问父类中被覆盖的方法，但可以在子类方法中调用父类中被覆盖的方法。如果需要在子类方法中调用父类中被覆盖的方法，则可以使用 super 作为调用者来调用父类中被覆盖的方法。

如果父类方法具有 private 访问权限，则该方法对其子类是隐藏的，因此其子类无法访问该方法，也就是无法重写该方法。如果子类中定义了一个与父类 private 方法具有相同的方法名、相同的形参列表、相同的返回值类型的方法，那么这不是重写，只是在子类中重新定义了一个新方法。例如，下面代码是完全正确的。

```
open class BaseClass {
    // test()方法具有 private 访问权限，子类不可访问该方法
    private fun test(){}
}
class SubClass: BaseClass(){
    // 此处并不是方法重写，所以不可以用 override 修饰
    fun test(){}
}
```

7.6.3 重写父类的属性

重写父类的属性与重写父类的方法大致相似：父类被重写的属性必须使用 open 修饰，子类重写的属性必须使用 override 修饰。此外，属性重写还有如下两个限制。

- 重写的子类属性的类型与父类属性的类型要兼容。
- 重写的子类属性要提供更大的访问权限。此处包含两方面的含义：①在访问权限方面，子类属性的访问权限应比父类属性的访问权限更大或相等。②只读属性可被读写属性重写，但读写属性不能被只读属性重写。

下面程序给出了重写父类属性的示例。

程序清单：codes\07\7.6\PropertyOverride.kt

```
open class Item{
    open protected var price: Double = 10.9
    open val name: String = ""
    open var validDays: Int = 0
}
class Book: Item{
    // 正确重写了父类属性，类型兼容，访问权限更大
    override public var price: Double
    // 正确重写了父类属性，读写属性重写只读属性
    override var name = "图书"
    // 重写错误，只读属性不能重写读写属性
    open val validDays: Int = 2
    constructor(){
```

```
        price = 3.0
    }
}
```

7.6.4 super 限定

如果需要在子类方法中调用父类中被覆盖的方法或属性，则可使用 super 限定。例如，为上面的 Ostrich 类添加一个方法，在这个方法中调用 Bird 类中被覆盖的 fly()方法。

```
fun callOverridedMethod(){
    // 在子类方法中通过 super 显式调用父类中被覆盖的方法
    super.fly()
}
```

借助 callOverridedMethod()方法，就可以让 Ostrich 对象既可以调用自己重写的 fly()方法，也可以调用 Bird 类中被覆盖的 fly()方法（调用 callOverridedMethod()方法即可）。

super 是 Kotlin 提供的一个关键字，用于限定该对象调用它从父类继承得到的属性或方法。

如果子类重写了父类的属性，那么子类中定义的方法直接访问该属性默认会访问到子类中定义的属性，无法访问到父类中被重写的属性。在子类定义的方法中可以通过 super 来访问父类中被重写的属性，如下面代码所示。

程序清单：codes\07\7.6\SuperTest.kt

```
open class BaseClass{
    open var a: Int = 5
}
class SubClass: BaseClass()
{
    override var a: Int = 7
    fun accessOwner(){
        println(a)
    }
    fun accessBase(){
        // 通过 super 限定访问从父类继承得到的 a 属性
        println(super.a)
    }
}
fun main(args: Array<String>) {
    val sc = SubClass()
    sc.accessOwner() // 输出 7
    sc.accessBase() // 输出 5
}
```

上面程序的 BaseClass 和 SubClass 中都定义了名为 a 的属性，则 SubClass 的 a 属性将会重写 BaseClass 的 a 属性。

程序中粗体字代码访问 super.a 时，使用 super 限定访问该对象从父类继承得到的 a 属性，而不是在当前类中定义的 a 属性。

如果子类中没有包含和父类同名的属性，那么在子类方法中访问该属性时，则无须显式使用 super 作为调用者。

如果在某个方法中访问名为 a 的属性，但没有显式指定调用者，则系统查找 a 的顺序为：

① 查找该方法中是否有名为 a 的局部变量。

② 查找当前类中是否包含名为 a 的属性。

③ 查找 a 的直接父类中是否包含名为 a 的属性，依次上溯查找 a 的所有父类，直到 Any 类，如果最终不能找到名为 a 的属性，则系统出现编译错误。

7.6.5 强制重写

如果子类从多个直接超类型（接口或类）继承了同名的成员，那么 Kotlin 要求子类必须重写该成员。如果需要在子类中使用 super 来引用超类型中的成员，则可使用尖括号加超类型名限定的 super 进行引用，如 super<Bar>。

如下程序示范了编译器强制重写的方法。

程序清单：codes\07\7.6\MandatoryOverride.kt

```
open class Foo {
    open fun test(){
        println("Foo 的 test")
    }
    fun foo(){
        println("foo")
    }
}
interface Bar {
    // 接口中成员默认是 open 的
    fun test(){
        println("Bar 的 test")
    }
    fun bar(){
        println("bar")
    }
}
class Wow : Foo(), Bar {
    // 编译器要求必须重写 test()
    override fun test(){
        super<Foo>.test() // 调用父类 Foo 的 test()
        super<Bar>.test() // 调用父接口 Bar 的 test()
    }
}
fun main(args: Array<String>){
    var w = Wow()
    w.test()
}
```

上面程序在 Foo 类中定义了一个可被重写的 test()方法，在 Bar 接口中也定义了一个可被重写的 test()方法（接口中的方法默认有 open 修饰符）。

当子类 Wow 同时继承 Foo、Bar 时，它会获得两个直接父类中定义的 test()方法，此时编译器要求子类 Wow 必须重写 test()方法，如上面程序的 Wow 类中的 test()方法所示。不仅如此，如果程序希望在子类中调用父类 Foo 的 test()方法，则可通过 super<Foo>.test()进行调用；如果希望在子类中调用父接口 Bar 的 test()方法，则可通过 super<Bar>.test()进行调用。

运行上面程序，可看到如下输出：

```
Foo 的 test
Bar 的 test
```

从上面输出可以看出，Wow 类的方法通过带尖括号标识的 super 成功调用了不同超类型

中定义的方法。

7.7 多态

与 Java 类似，Kotlin 的变量也有两个类型：一个是编译时类型，一个是运行时类型。编译时类型由声明该变量时使用的类型决定，运行时类型由实际赋给该变量的对象决定。如果编译时类型和运行时类型不一致，就可能出现所谓的多态（Polymorphism）。

7.7.1 多态性

先看下面程序。

程序清单：codes\07\7.7\SubClass.kt

```
open class BaseClass{
    open var book = 6
    fun base(){
        println("父类的普通方法")
    }
    open fun test(){
        println("父类的被覆盖的方法")
    }
}
class SubClass: BaseClass(){
    // 重写父类的属性
    override var book = 60
    // 重写父类的方法
    override fun test(){
        println("子类的覆盖父类的方法")
    }
    fun sub(){
        println("子类的普通方法")
    }
}
fun main(args: Array<String>) {
    // 下面编译时类型和运行时类型完全一样，因此不存在多态
    var bc: BaseClass = BaseClass()
    // 输出 6
    println(bc.book)
    // 下面两次调用将执行 BaseClass 的方法
    bc.base()
    bc.test()
    // 下面编译时类型和运行时类型完全一样，因此不存在多态
    var sc: SubClass = SubClass()
    // 输出 60
    println(sc.book)
    // 下面调用将执行从父类继承的 base()方法
    sc.base()
    // 下面调用将执行当前类的 test()方法
    sc.test()
    // 下面编译时类型和运行时类型不一样，多态发生
    var ploymophicBc: BaseClass = SubClass()
    // 输出 60 —— 表明访问的依然是子类对象的属性
    println(ploymophicBc.book)
    // 下面调用将执行从父类继承的 base()方法
```

```
    ploymophicBc.base()
    // 下面调用将执行当前类的 test()方法
    ploymophicBc.test()
    // 因为 ploymophicBc 的编译时类型是 BaseClass
    // BaseClass 类没有提供 sub()方法，所以下面代码编译时会出现错误
//    ploymophicBc.sub()
}
```

上面程序的 main()函数中显式创建了 3 个变量，对于前两个变量 bc 和 sc，它们的编译时类型和运行时类型完全相同，因此调用它们的属性和方法非常正常，完全没有任何问题。但第三个变量 ploymophicBc 则比较特殊，它的编译时类型是 BaseClass，而运行时类型是 SubClass，当调用该变量的 test()方法（BaseClass 类中定义了该方法，子类 SubClass 覆盖了父类的该方法）时，实际执行的是 SubClass 类中覆盖后的 test()方法，这就可能出现多态。

因为子类其实是一种特殊的父类，因此 Kotlin 允许把一个子类对象直接赋给一个父类变量，无须任何类型转换，或者被称为向上转型（upcasting），向上转型由系统自动完成。

当把一个子类对象直接赋给父类变量时，例如上面的 var ploymophicBc: BaseCalss = new SubClass()，这个 ploymophicBc 变量的编译时类型是 BaseClass，而运行时类型是 SubClass，当运行时调用该变量的方法时，其方法行为总是表现出子类方法的行为特征，而不是父类方法的行为特征，这就可能出现：相同类型的变量，调用同一个方法时呈现出多种不同子类的行为特征，这就是多态。

上面的 main()函数中注释掉了 ploymophicBc.sub()，这行代码会在编译时引发错误。虽然 ploymophicBc 引用的对象实际上确实包含 sub()方法（例如，可以通过反射来执行该方法），但因为它的编译时类型为 BaseClass，因此编译时无法调用 sub()方法。

与方法类似的是，对象的属性同样具有多态性。比如上面通过 ploymophicBc 输出它的 book 属性变量时，依然是输出 SubClass 类中定义的属性，而不是输出 BaseClass 父类中定义的属性。

注意：

变量在编译阶段只能调用其编译时类型所具有的方法，但运行时则执行其运行时类型所具有的方法。因此，在编写 Kotlin 代码时，变量只能调用声明它时所用的类包含的方法。例如，通过 var p: Any = new Person()代码定义一个变量 p，则这个 p 只能调用 Any 类的方法，而不能调用 Person 类中定义的方法。



7.7.2 使用 is 检查类型

变量只能调用其编译时类型的方法，而不能调用其运行时类型的方法，即使它实际所引用的对象确实包含该方法。如果要让这个变量调用其运行时类型的方法，就需要把它强制转换成运行时类型，强制转换需要借助于强制转型运算符。Kotlin 的类型转换运算符包含 as 和 as?两个。

由于向上转型可由 Kotlin 自动转换，因此强制转型通常总是向下转型。

为了保证类型转换不会出错，Kotlin 提供了类型检查运算符：is 和!is。

is 运算符的前一个操作数通常是一个变量，后一个操作数通常是一个类（也可以是接口，可以把接口理解成一种特殊的类），它用于判断前面的变量是否引用后面的类，或者其子类、实现类的实例。如果是，则返回 true，否则返回 false。



在使用 is 运算符时需要注意，is 运算符前面操作数的编译时类型要么与后面的类相同，要么与后面的类具有父子继承关系，否则编译时程序报错。



!is 就是 is 运算符的反义词。

下面程序示范了 is 运算符的用法。

程序清单：codes\07\7.7\IsTest.kt

```
import java.util.Date
// 定义一个接口
interface Testable{}
fun main(args: Array<String>) {
    // 声明 hello 时使用 Any 类，则 hello 的编译时类型是 Any
    // hello 变量的实际类型是 String
    val hello: Any = "Hello"
    println("字符串是否是 String 类的实例：${hello is String}") // 返回 true
    // Date 与 Any 类存在继承关系，可以进行 is 运算
    println("字符串是否是 Date 类的实例：${hello is Date}")  // 返回 false
    // String 没有实现 TestProtocol 协议，所以返回 false
    println("字符串是否是 Testable 协议的实例：${hello is Testable}")  // 返回 false
    val a: String = "Hello"
    // String 类与 Date 类没有继承关系，所以下面代码编译出现错误
    println("字符串是否是 Date 类的实例：${a is Date}")
}
```

上面程序通过 val hello: Any = "Hello"代码定义了一个 hello 变量，这个变量的编译时类型是 Any 类，但实际类型是 String。因为 Any 类是所有 Kotlin 类的父类，因此可以执行 hello is String 和 hello is Date 等。

但使用 val a: String = "Hello"代码定义的变量 a，就不能执行 a is Date，因为 a 的编译时类型是 String，String 与 Date 之间没有继承关系，所以这行代码导致编译错误。

is 运算符的作用是，在进行强制类型转换之前，首先判断前一个变量是否引用后一个类，或者其子类的实例，是否可以成功转换，从而保证代码更加健壮。

此外，Kotlin 的 is 和!is 都非常智能，只要程序使用 is 或!is 对变量进行了判断，系统就会自动将变量的类型转换为目标类型。例如如下代码。

程序清单：codes\07\7.7\IsTest2.kt

```
fun main(args: Array<String>) {
    var a: Any = "fkit"
    // 直接访问 a 的 length 属性，编译器报错
    // 由于 a 的编译时类型是 Any，因此编译时它没有 length 属性
//    println(a.length)
    // 先判断 a 为 String，在 if 条件体中 a 被自动转换为 String 类型
    if (a is String) println(a.length)
}
```

上面变量 a 的编译时类型是 Any，对于编译器而言，它只能被当成 Any 类型使用，因此上面程序中第一行粗体字代码访问 a 的 length 属性会导致编译错误；对于第二行粗体字代码，程序先执行 if(a is String)判断，如果该判断为 true，系统可进入执行 if 条件体，在 if 条件体中

变量 a 被自动转型为 String。

!is 同样有这种智能转换的功能。例如如下函数（程序清单同上）：

```
fun test(x: Any){
    // 如果 x 不是 String，函数返回
    if (x !is String) return
    // 因此以下部分，x 会被自动转换为 String 类型
    println("x 的长度为：${x.length}")
}
```

使用 when 分支进行判断也可自动转换。例如如下代码（程序清单同上）：

```
when(a) {
    // 如果进入该分支，则表明 a 是 String 类型，可调用 String 的方法
    is String -> println(a.length)
    // 如果进入该分支，则表明 a 是 Int 类型，可调用 Int 的方法
    is Int -> println(a.toDouble())
}
```

总之，当程序使用 is 或!is 判断之后，只要 Kotlin 能推断出变量属于某种类型，Kotlin 就会自动将该变量的编译时类型转换为指定类型。比如使用&&运算符也可以。例如如下代码（程序清单同上）：

```
// 能进入&&之后的表达式，表明 a 是 String 类型
// 因此在&&之后可调用 a 的 length 属性
if (a is String && a.length > 3) {
    println("a 的长度大于 3")
}
```

用||运算符也行。例如如下函数（程序清单同上）：

```
fun foo(x: Any){
    // 能进入||之后的表达式，表明 a 是 String 类型
    // 因此在||之后可调用 a 的 length 属性
    if (x !is String || x.length == 0)
        return
    // 以下部分，x 会被自动转换为 String 类型
    println("x 的长度为：${x.length}")
}
```

7.7.3 使用 as 运算符转型

除使用 is 自动转型之外，Kotlin 也支持使用 as 运算符进行强制转型。Kotlin 提供了两个向下转型运算符。

- as：不安全的强制转型运算符，如果转型失败，程序将会引发 ClassCastException 异常。
- as?：安全的强制转型运算符，如果转型失败，程序不会引发异常，而是返回 null。

强制向下转型只能在具有继承关系的两种类型之间进行，如果是两种没有任何继承关系的类型，则无法进行类型转换，编译时就会发出警告：转换不可能成功。如果试图把一个父类变量转换成子类类型，则该变量实际引用的实例必须是子类实例才行（即编译时类型为父类类型，而运行时类型是子类类型），否则就会转换失败。

下面是进行强制类型转换的示范程序。下面程序详细说明了哪些情况可以进行类型转换，哪些情况不可以进行类型转换。

程序清单：codes\07\7.7\ConversionTest.kt

```
fun main(args: Array<String>) {
    val obj: Any = "Hello"
    // obj 变量的编译时类型为 Any，Any 与 String 存在继承关系，可以进行转换
    // 而且 obj 实际引用的实例是 String 类型，所以运行时也可通过
    val objStr = obj as String
    println(objStr)
    // 定义一个 objPri 变量，编译时类型为 Any，实际类型为 Int
    val objPri: Any = 5
    // objPri 变量的编译时类型为 Any，objPri 的运行时类型为 Int
    // Any 与 String 存在继承关系，可以进行转换，编译通过
    // 但 objPri 变量实际引用的实例是 Int 类型，所以转换失败
    val str: String = objPri as String  // 转换失败
    val fkit = "fkit.org"
    val s: String = "Kotlin"
    // s 变量的编译时类型为 String，s 的运行时类型为 String
    // 但 String 与 Number 不存在继承关系，因此编译发出警告：转换不可能成功
    val num: Number = s as Number  // 警告：转换不可能成功
}
```

注意：

当把子类实例赋给父类变量时，被称为向上转型（upcasting），这种转型总是可以成功的，这也从另一个侧面证实了子类是一种特殊的父类。这种转型只是表明这个变量的编译时类型是父类，但实际执行它的方法时，依然表现出子类的行为方式。当把一个父类变量赋给子类变量时，就需要进行向下转型，向下转型必须使用 as?或 as 运算符，向下转型可能失败。



使用 as 执行的强制转型是不安全的转换，如果强制转型失败则会引发 ClassCastException 异常，比如上面程序会在第二行粗体字代码处引发该异常。此时可用 as?执行安全的类型转换，使用 as?进行转换时，如果转换失败则会返回 null。

由于 as?转换返回的是可空的值，因此程序需要对 as?转换的结果进行 null 判断。例如如下程序。

程序清单：codes\07\7.7\SafeConversion.kt

```
open class Fruit {
    var name: String
    var weight: Double
    constructor(name: String, weight: Double) {
        this.name = name
        this.weight = weight
    }
}
class Apple: Fruit {
    var color : String
    constructor(name:String, weight:Double, color:String):super(name, weight){
        this.color = color
    }
}
class Grape: Fruit {
    var sugarRate: Double
    constructor(name: String, weight: Double, sugarRate: Double):super(name, weight){
```

```
        this.sugarRate = sugarRate
    }
}
fun main(args: Array<String>) {
    // 使用数组保存 4 个水果
    var fruits = arrayOf(
        Apple("红富士", 1.8, "粉红"),
        Apple("花牛果", 2.3, "红色"),
        Grape("巨峰", 1.4, 0.34),
        Grape("加州提子", 2.2, 0.45))
    // 由于 fruits 的类型是 Array<Fruit>，因此程序只知道该数组元素是 Fruit
    for (f in fruits) {
        // 此处使用 as?强制转型，因此 ap 的类型是 Apple?
        var ap = f as? Apple
        // 由于 ap 的类型是 ap?，因此程序使用?.语法来访问它的属性
        println("${ap?.name}苹果的颜色是：${ap?.color}")
    }
}
```

上面程序向 fruits 数组（类型为 Array<Fruit>）中添加了 4 个元素，程序只知道该数组元素是 Fruit 的实例，但并不清楚具体是哪个子类的实例。

程序的末尾使用 for-in 循环遍历了 fruits 数组中的元素，遍历时使用 as?将数组元素强制转换为 Fruit 的子类，由于 as?返回的是可空类型，因此程序还使用了?.语法来访问可空类型的属性值。

编译、运行该程序，可以看到如下输出：

```
红富士苹果的颜色是：粉红
花牛果苹果的颜色是：红色
null 苹果的颜色是：null
null 苹果的颜色是：null
```

7.8 本章小结

本章主要介绍了 Kotlin 面向对象的基本知识，包括如何定义类，如何为类定义属性、方法，以及如何创建类的对象。方法也是本章介绍的重点，本章详细介绍了方法的参数传递机制、中缀表示法、componentN 方法和解构。本章还详细介绍了 Kotlin 的属性机制，属性相当于 Java 的 field、getter 和 setter 方法的组合，因此用起来非常方便。

本章详细讲解了如何使用包、访问控制符来设计封装良好的类，读者需要认真学习 Kotlin 的 private、internal、protected 和 public 四个访问控制符，并理解 Kotlin 的访问控制符的默认规则。

本章还重点讲解了 Java 的继承和多态，包括 Kotlin 实现继承的语法、子类构造器委托父类的各种方式，以及把一个子类对象赋给父类变量时产生的多态行为。

CHAPTER

8

第8章 面向对象（下）

本章要点

- 理解 Kotlin 扩展的实现机制
- 扩展方法和扩展属性
- 以成员方式声明扩展
- 带接收者的匿名函数
- const 常量
- final 属性和 final 方法
- 不可变类和可变类
- abstract 关键字的用法
- 实现模板模式
- 定义接口的语法
- 实现接口
- 接口和抽象类的联系与区别
- 嵌套类和内部类
- 创建嵌套类和内部类的对象
- 为嵌套类派生子类
- 对象表达式
- 对象声明和单例模式
- 伴生对象和静态成员
- 伴生对象的扩展
- Kotlin 的枚举类
- 枚举类的成员变量、方法和构造器
- 实现接口的枚举类
- 包含抽象方法的枚举类
- 类委托
- 属性委托
- 延迟属性、属性监听、Map 存储属性值
- 局部属性委托

本章将会继续介绍 Kotlin 面向对象的高级部分。扩展是 Kotlin 提供的一种非常优秀的机制，扩展机制也是很多现代编程语言都支持的功能，而 Kotlin 通过扩展可以弥补 Java 作为静态语言灵活性不足的问题。

Kotlin 默认为所有类、方法、属性都提供了 final 关键字来修饰，这意味着在默认情况下，Kotlin 的类不可派生子类，Kotlin 的方法、属性都不可被重写。为了取消默认的 final 修饰符，Kotlin 提供了 final 的反义词：open，本章将会详细介绍这两个修饰符。

abstract 和 interface 两个关键字分别用于定义抽象类和接口，抽象类和接口都是从多个子类中抽象出来的共同特征。但抽象类主要作为多个类的模板，而接口则定义了多个类应该遵守的规范。

与 Java 的内部类相似，Kotlin 提供了嵌套类和内部类，其中嵌套类相当于 Java 的静态内部类；而内部类则相当于 Java 的非静态内部类，因此本书将会结合 Java 特征来介绍 Kotlin 的嵌套类和内部类。

Kotlin 提供了对象表达式来代替 Java 的匿名内部类语法，其实对象表达式相当于增强版的匿名内部类语法：匿名内部类只能指定一个父类型，而对象表达式则可指定多个父类型。此外，本章还将介绍对象声明和伴生对象，其中对象声明可以非常方便地实现单例模式，而伴生对象则可弥补 Kotlin 不支持静态成员的不足。

enum class 关键字用于创建枚举类，枚举类是一种不能自由创建对象的类，枚举类的对象在定义类时已经固定下来。枚举类特别适合定义像行星、季节这样的类，它们能创建的实例是有限且确定的。本章最后还会介绍 Kotlin 的类委托和属性委托机制，通过属性委托，Kotlin 可实现延迟属性、属性监听等其他现代编程语言普遍支持的功能。

8.1 扩展

Kotlin 的扩展是一个很独特的功能，Java 本身并不支持扩展，Kotlin 为了让扩展能在 JVM 平台上运行，必须做一些独特的处理。本节将会详细介绍这些处理。

Kotlin 支持扩展方法和扩展属性。

8.1.1 扩展方法

扩展方法的语法很简单，其实就是定义一个函数，只是函数名不要写成简单的函数，而是要在函数名前增加被扩展的类（或接口）名和点号（.）。例如如下程序。

程序清单：codes\08\8.1\Row.kt

```
open class Raw{
    fun test(){
        println("test 方法")
    }
}
```

上面程序定义了一个 Raw 类，该 Raw 类使用了 open 修饰符，这是为了示范给 Raw 类派生子类。如果只是对 Raw 类进行扩展，那么 Kotlin 并不需要使用 open 修饰该类。

接下来，为 Raw 类派生如下类，代码如下。

程序清单：codes\08\8.1\RowSub.kt

```
class RawSub: Raw(){
    fun sub(){
        println("--sub()方法--")
    }
}
```

上面 RawSub 类继承了 Raw 类，并增加了一个新方法：sub()。

接下来，程序为 Raw 类扩展一个方法，代码如下。

程序清单：codes\08\8.1\Row_info.kt

```
// 定义一个函数，函数名为“被扩展类.方法”
fun Raw.info(){
    println("===扩展的 info 方法===")
}
```

程序为 Raw 类扩展了 info()方法之后，就像为 Raw 类增加了 info()方法一样，所有的 Raw 对象都可调用 info()方法。不仅如此，Raw 类的子类的实例也可调用 info 方法。例如如下程序。

程序清单：codes\08\8.1\ExtensionTest.kt

```
fun main(args: Array<String>) {
    var t = Raw()
    t.test()
    // 调用 Raw 对象扩展的方法
    t.info()
    // 创建 Raw 类的子类的实例
    var rs = RawSub()
    rs.test()
    rs.sub()
    // Raw 的子类的对象也可调用扩展的方法
    rs.info()
}
```

从上面粗体字代码可以看出，不管是 Raw 类的实例，还是 Raw 子类的实例，都可调用我们为 Raw 类扩展的方法，就像真的为 Raw 类新增了一个方法一样。

扩展方法中的 this 与成员方法中的 this 一样，也代表调用该方法的对象。

当然，扩展也可以为 Kotlin 系统提供的类增加方法。例如，下面程序为系统提供的 List 集合扩展了一个 shuffle()方法，该方法用于对集合元素进行随机排列。

程序清单：codes\08\8.1\List_shuffle.kt

```
// 该方法的实现思路是：先生成 List 集合所有索引的随机排列
// 然后根据随机排列的索引去 List 集合中取元素
fun List<Int>.shuffle(): List<Int>{
    val size = this.size
    // 下面的 indexArr 用于保存 List 集合的索引的随机排列
    var indexArr = Arr<Int>(size, {0})
    var result: MutableList<Int> = mutableListOf()
    // 创建随机对象
    val rand = java.util.Random()
    var i = 0
    outer@ while (i < size ) {
        // 生成随机数
        var r = rand.nextInt(size)
```

```
        for (j in 0 until i) {
            // 如果 r 和前面已生成的任意数字相等，则该随机数不可用，需要重新生成
            if (r == indexArr[j]) {
                continue@outer
            }
        }
        // 如果上面的循环结束了都没有执行 continue，则说明该 r 是一个不重复的随机数
        // 将随机数 r 存入 indexArr 数组中
        indexArr[i] = r
        // 根据随机的索引读取 List 集合元素，并将元素添加到 result 集合中
        result.add(this[r])
        i++
    }
    return result.toList()
}
```

上面的粗体字代码看似定义了一个名为List<Int>.shuffle的函数，而实际上就是为List<Int>扩展了一个shuffle()方法。如下程序示范了调用List集合的shuffle()方法。

程序清单：codes\08\8.1\ListTest.kt

```
fun main(args: Array<String>) {
    var nums = listOf(20, 30, 100, 34, 67)
    // 调用程序为 List 扩展的 shuffle()方法
    println(nums.shuffle())
    println(nums.shuffle())
}
```

此外，上面程序扩展List时指定了它的泛型形参为Int。也就是说，该扩展仅对List<Int>有效，因此List<String>就不能调用shuffle()方法。

实际上Kotlin完全支持直接进行泛型类扩展，只不过此时需要在函数上使用泛型，因此要使用泛型函数的语法。例如如下代码。

程序清单：codes\08\8.1\List_shuffle2.kt

```
// 该方法的实现思路是：先生成 List 集合所有索引的随机排列
// 然后根据随机排列的索引去 List 集合中取元素
fun <T> List<T>.shuffle(): List<T>{
    val size = this.size
    // 下面的 indexArr 用于保存 List 集合的索引的随机排列
    var indexArr = Array<Int>(size, {0})
    var result: MutableList<T> = mutableListOf()
    // 创建随机对象
    val rand = java.util.Random()
    var i = 0
    outer@ while (i < size ) {
        // 生成随机数
        var r = rand.nextInt(size)
        for (j in 0 until i) {
            // 如果 r 和前面已生成的任意数字相等，则该随机数不可用，需要重新生成
            if (r == indexArr[j]) {
                continue@outer
            }
        }
        // 如果上面的循环结束了都没有执行 continue，则说明该 r 是一个不重复的随机数
        // 将随机数 r 存入 indexArr 数组中
        indexArr[i] = r
```

```
            // 根据随机的索引读取 List 集合元素，并将元素添加到 result 集合中
            result.add(this[r])
            i++
        }
        return result.toList()
    }
```

请留意上面函数签名中的 fun 关键字和函数名之前的<T>声明，其用于声明一个泛型函数。这样，程序就为 List<T>扩展了一个方法：shuffle()，List<Int>、List<String>等集合都可调用该方法。例如，在上面的 ListTest.kt 程序中增加如下代码：

```
var strList = listOf("Kotlin", "Java", "Go", "Erlang")
println(strList.shuffle())
println(strList.shuffle())
```

8.1.2 扩展的实现机制

我们知道，Java 是一门静态语言。一个类被定义完成之后，程序无法动态地为该类增加、删除成员（field、方法等），除非开发者重新编辑该类的源代码，并重新编译该类。

但现在 Kotlin 的扩展却好像可以动态地为一个类增加新的方法，而且不需要重新修改该类的源代码，这真是太神奇了！那 Kotlin 扩展的实际情况是怎样的呢？难道 Kotlin 可以突破 JVM 的限制？

实际上，Kotlin 的扩展并没有真正地修改所扩展的类，被扩展的类还是原来的类，没有任何改变。Kotlin 扩展的本质就是定义了一个函数，当程序用对象调用扩展方法时，Kotlin 在编译时会执行静态解析——就是根据调用对象、方法名找到扩展函数，转换为函数调用。例如，上面程序中的如下代码：

```
strList.shuffle()
```

Kotlin 在编译时这行代码按如下步骤执行。

① 检查 strList 的类型，发现其类型是 List<String>。

② 检查 List<String>类本身是否定义了 shuffle()方法，如果该类本身包含该方法，则 Kotlin 无须进行处理，直接编译即可。

③ 如果 List<String>类本身不包含 shuffle()方法，则 Kotlin 会查找程序是否为 List<String>扩展了 shuffle()方法——也就是查找系统中是否包含了名为 List<String>.shuffile()的函数（或泛型函数）定义，如果找到该函数，则 Kotlin 编译器会执行静态解析，它会将上面代码替换成执行 List<String>.shuffile()函数。

④ 如果 List<String>不包含 shuffle()方法，也找不到名为 List<String>.shuffile()的函数（或泛型函数）定义，编译器将报错。

由此可见，Kotlin 的扩展并没有真正改变被扩展的类，Kotlin 只是耍了一个小“花招”，当 Kotlin 程序调用扩展方法时，Kotlin 编译器会将这行代码静态解析为调用函数，这样 JVM 就可接受了。这意味着调用扩展方法是由其所在表达式的编译时类型决定的，而不是由它所在表达式的运行时类型决定的。

看如下示范代码。

程序清单：codes\08\8.1\ExtensionParse.kt

```
// 定义一个 Base 类
open class Base
```

```
// 定义 Sub 类继承 Base 类
class Sub: Base()
// 为 Base 类扩展 foo 方法
fun Base.foo() = println("Base 扩展的 foo()方法")
// 为 Sub 类扩展 foo 方法
fun Sub.foo() = println("Sub 扩展的 foo()方法")
// 定义一个函数
fun invokeFoo(b: Base) {
    // 调用 Base 对象的 foo()方法
    b.foo()
}
fun main(args: Array<String>) {
    // 传入的是 Sub 对象
    invokeFoo(Sub())  // ①
}
```

上面程序中定义了一个 Base 类及其子类 Sub，接下来程序为 Base 和 Sub 分别扩展了 foo()方法。

对于有继承关系的 Base 和 Sub 两个类，它们都包含了具有相同签名的 foo()方法，如果 foo()方法是成员方法，程序就必须用 override 声明子类方法重写父类方法；但由于此处采用的是扩展，因此 Kotlin 不需要声明为方法重写。

上面程序中定义了一个 invokeFoo(Base)函数，该函数的形参类型是 Base，但程序在①号粗体字代码处调用该函数时传入一个 Sub 对象，那么此处到底是调用 Base 类扩展的 foo()方法，还是调用 Sub 类扩展的 foo()方法呢？

如果 foo()方法是 Base、Sub 所包含的成员方法（方法重写），那么一定是由 JVM 动态解析为调用运行时类型（Sub 对象）的方法，这一点在前一章中介绍多态时讲得很清楚。

但此处的 foo()方法是 Base、Sub 所包含的扩展方法，对于扩展方法由于 Kotlin 编译器执行静态解析，在编译阶段 Kotlin 编译器只知道 invokeFoo()方法的形参是 Base 类型，因此 Kotlin 编译器会将其替换为调用 Base 的 foo()方法。

运行上面程序，可看到如下输出：

```
Base 扩展的 foo()方法
```

注意：

总结起来一句话：成员方法执行动态解析（由运行时类型决定）；扩展方法执行静态解析（由编译时类型决定）。



如下两行代码同样示范了这种效果（程序清单同上）：

```
val bb: Base = Sub()
bb.foo()
```

上面代码定义了一个 bb 变量，其编译时类型是 Base，运行时类型是 Sub，接下来程序调用了 bb 的 foo()方法，读者需要判断 foo()方法到底是成员方法，还是扩展方法。如果 foo()是成员方法，那么 foo()方法表现出运行时类型的方法行为；如果 foo()是扩展方法，那么 foo()方法总是表现出编译时类型的方法行为。

此外，前面介绍了 Kotlin 编译时解析一个成员方法和扩展方法的步骤，由此可知成员方法的优先级高于扩展方法。这意味着，如果一个类包含了具有相同签名的成员方法和扩展方法，

当程序调用这个方法时，系统总是会执行成员方法，而不会执行扩展方法。例如如下程序。

程序清单：codes\08\8.1\ExtensionAndMember.kt

```
class ExtensionAndMember {
    // 为该类定义成员方法：foo()
    fun foo() = println("成员方法")
}
// 为ExtensionAndMember类定义扩展方法：foo()
fun ExtensionAndMember.foo() = println("扩展方法")
fun main(args: Array<String>){
    var ea = ExtensionAndMember()
    // ea调用foo()方法，优先使用成员方法
    ea.foo() // 输出"成员方法"
}
```

上面程序中粗体字代码定义了和成员方法具有相同签名的扩展方法，该方法的优先级将总是低于具有相同签名的成员方法，因此编译器会在该行提示如下警告：

```
warning: extension is shadowed by a member: public final fun foo(): Unit
```

上面警告提示：扩展方法被成员方法遮蔽。

但如果扩展方法和成员方法只是具有相同的方法名，但它们的形参列表不同，这种情况就不算遮蔽，因为它们其实是两个方法，Kotlin也可正常区分它们。例如，我们可以为上面的类扩展如下foo()方法，它们都不会发生遮蔽。

```
fun ExtensionAndMember.foo(name: String) = println("扩展方法,参数:${name}")
fun ExtensionAndMember.foo(age: Int) = println("扩展方法,参数:${age}")
```

扩展与类、接口一样，如果扩展定义在其他包中，一样也需要使用import进行导包。

在网络上看到有人问，Java是否可以调用Kotlin类的扩展方法呢？乍一看似乎不能，因为Java本身并不支持扩展，但如果真正理解了Kotlin扩展的实现机制，就不难发现使用Java调用扩展方法非常简单：调用扩展方法的本质就是静态解析成调用扩展函数。下面代码示范了用Java调用扩展方法。

程序清单：codes\08\8.1\InvokeExtension.java

```
public class InvokeExtension{
    public static void main(String[] args){
        // 创建Raw对象
        Raw t = new Raw();
        // 调用Raw对象的成员方法
        t.test();
        // 调用Raw对象的扩展方法，需要自己解析成调用扩展函数
        Raw_infoKt.info(t);
        // 创建RawSub对象
        RawSub rs = new RawSub();
        // 调用RawSub对象的成员方法
        rs.test();
        rs.sub();
        // 调用RawSub对象的扩展方法，需要自己解析成调用扩展函数
        Raw_infoKt.info(rs);
    }
}
```

通过上面两行粗体字代码，即可在 Java 程序中调用 Kotlin 扩展的方法。结合此处用 Java 调用 Kotlin 扩展方法的代码，我们能更好地理解 Kotlin 扩展的静态解析机制。

在 Java 代码中调用 Kotlin 扩展方法与 Kotlin 本身调用扩展方法的区别在于：Kotlin 编译器本身支持静态解析，因此程序可直接用对象调用扩展方法；而 Java 编译器本身不支持静态解析，因此需要开发者写代码时就做好静态解析。

8.1.3 为可空类型扩展方法

Kotlin 还允许为可空类型（带“?”后缀的类型）扩展方法。由于可空类型允许接受 null 值，这样使得 null 值也可调用该扩展方法。从另一方面来看，由于会导致 null 值调用该扩展方法，因此程序需要在扩展方法中处理 null 值的情形。例如如下程序。

程序清单：codes\08\8.1\NullableExtension.kt

```
// 为可空类型扩展 equals 方法
fun Any?.equals(other: Any?): Boolean{
    if (this == null) {
        return if (other == null ) true else false
    }
    return other.equals(other)
}
fun main(args: Array<String>) {
    var a = null
    println(a.equals(null)) // 输出 true
    println(a.equals("Kotlin")) // 输出 false
}
```

上面粗体字代码示范了为可空类型扩展 equals()方法，这样 null 值也可调用 equals()方法与其他对象进行比较，如上面程序的 main()函数中代码所示。

8.1.4 扩展属性

Kotlin 也允许扩展属性，但由于 Kotlin 的扩展并不能真正修改目标类，因此 Kotlin 扩展的属性其实是通过添加 getter、setter 方法实现的，没有幕后字段。简单来说，扩展的属性只能是计算属性！

由于 Kotlin 的扩展属性只能是计算属性，因此对扩展属性有如下两个限制。

- 扩展属性不能有初始值（没有存储属性值的幕后字段）。
- 不能用 field 关键字显式访问幕后字段。
- 扩展只读属性必须提供 getter 方法；扩展读写属性必须提供 getter、setter 方法。

如下程序示范了为 Kotlin 类添加扩展属性。

程序清单：codes\08\8.1\ExtensionProperty.kt

```
class User(var first:String, var last: String){
}
// 为 User 扩展读写属性
var User.fullName: String
    get() = "${first}.${last}"
    set(value) {
        println("执行扩展属性 fullName 的 setter 方法")
        // value 字符串中不包含.或包含几个.都不行
        if ("." !in value || value.indexOf(".") != value.lastIndexOf(".")) {
```

```
            println("您输入的 fullName 不合法")
        } else {
            var tokens = value.split(".")
            first = tokens[0]
            last = tokens[1]
        }
    }
fun main(args: Array<String>) {
    var user = User("悟空", "孙")
    println(user.fullName)
    user.fullName = "八戒.猪"
    println(user.first)
    println(user.last)
}
```

上面程序中粗体字代码为 User 类扩展了一个 fullName 读写属性，扩展该属性时重写了 getter、setter 方法。

此外，由于扩展属性的本质就是 getter、setter 方法，因此也可用泛型函数的形式来定义扩展属性。例如，Kotlin 为 List 扩展的 lastIndex 属性的代码如下：

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

上面的扩展属性在 var 和 List<T>之间定义了泛型形参，这是典型的泛型函数的语法。

8.1.5 以成员方式定义扩展

前面见到的扩展，都是以顶层函数的形式（放在包空间下）进行定义的，因此这些扩展都可直接使用（如果扩展位于不同的包中，当然也需要导包）。Kotlin 还支持以类成员的方式定义扩展——就像为类定义方法、属性那样定义扩展。

对于以类成员方式定义的扩展，一方面它属于被扩展的类，因此在扩展方法（属性）中可直接调用被扩展类的成员（省略 this 前缀）；另一方面它又位于定义它所在类的类体中，因此在扩展方法（属性）中又可直接调用它所在类的成员（省略 this 前缀）。

例如如下程序。

程序清单：codes\08\8.1\ExtensionInClass.kt

```
class A{
    fun bar() = println("A 的 bar 方法")
}
class B{
    fun baz() = println("B 的 baz 方法")
    // 以成员方式为 A 扩展 foo()方法
    fun A.foo(){
        // 在该方法内既可调用类 A 的成员，也可调用类 B 的成员
        bar() // A 对象为隐式调用者
        baz() // B 对象为隐式调用者
    }
    fun test(target: A){
        // 调用 A 对象的成员方法
        target.bar()
        // 调用 A 对象的扩展方法
        target.foo()
    }
}
```

```
fun main(args: Array<String>) {
    var b = B()
    b.test(A())
}
```

上面程序在类 B 中为类 A 扩展了一个 foo()方法。由于该 foo()方法一方面属于类 A，另一方面又定义在类 B 中，因此在 foo()方法内既可直接调用类 A 的成员，也可直接调用类 B 的成员，且都不需要使用 this 前缀，如上面程序中粗体字代码所示。

这样又会产生一个新的问题。如果被扩展类和扩展定义所在的类包含了同名的方法，此时就会导致：程序在扩展方法中调用两个类都包含的方法时，系统总是优先调用被扩展类的方法。为了让系统调用扩展定义所在类的方法，必须使用带标签的 this 进行限定。

例如如下程序。

程序清单：codes\08\8.1\Distinct.kt

```
class Tiger{
    fun foo(){
        println("Tiger 类的 foo()方法")
    }
}
class Bear{
    fun foo(){
        println("Bear 类的 foo()方法")
    }
    // 以成员方式为 Tiger 类扩展 test()方法
    fun Tiger.test(){
        foo()
        // 使用带标签的 this 指定调用 Bear 的 foo()方法
        this@Bear.foo()
    }
    fun info(tiger: Tiger){
        tiger.test()
    }
}
fun main(args: Array<String>) {
    val b = Bear()
    b.info(Tiger())
}
```

上面程序在 Bear 类中为 Tiger 类扩展了一个 test()方法，且 Bear 类和 Tiger 类都包含 foo()方法，因此如果程序在 test()扩展方法中调用 foo()方法，系统总是优先调用 Tiger 类的 foo()方法。为了在 test()扩展方法中调用 Bear 定义的 foo()方法，就需要使用 this@Bear 前缀，如上面程序中粗体字代码所示。

Kotlin 的 this 比 Java 的 this 更强大，Kotlin 的 this 支持用“@类名”形式，这种形式限制了该 this 代表哪个类的对象。

8.1.6 带接收者的匿名函数

Kotlin 还支持为类扩展匿名函数，在这种情况下，该扩展函数所属的类也是该函数的接收者。因此，这种匿名函数也被称为“带接收者的匿名函数”。

与普通扩展方法不同的是：去掉被扩展类的类名和点（.）之后的函数名即可，其他部分并没有太大的区别。与普通扩展方法相似的是，带接收者的匿名函数（相当于扩展匿名函数）

也允许在函数体内访问接收者对象的成员。

例如如下程序。

程序清单：codes\08\8.1\ExtensionAnnoymous.kt

```
// 定义一个带接收者的匿名函数
val factorial = fun Int.(): Int{
    // 该匿名函数的接收者是 Int 对象
    // 因此在该匿名函数中，this 代表调用该匿名函数的 Int 对象
    if (this < 0) {
        return -1
    } else if(this == 1){
        return 1
    } else {
        var result = 1
        for (i in 1 .. this ) {
            result *= i
        }
        return result
    }
}
fun main(args: Array<String>) {
    println(6.factorial())
}
```

上面程序中定义了一个带接收者的匿名函数，注意第一行粗体字代码，相当于程序为 Int 扩展了一个匿名函数。如果写成 fun Int.abc(): Int，就表示为 Int 类扩展了 abc()方法；但是 fun Int.(): Int，此时 Int.后什么也没有，也就是没有指定函数名，因此它是一个匿名函数。

由于上面程序最后将带接收者的匿名函数赋值给了 factorial 变量，因此可通过 Int 对象来调用 factorial()函数，如上面程序中最后一行粗体字代码所示。

与普通函数相似的是，带接收者的匿名函数也有自身的类型，即带接收者的函数类型。例如，上面 factorial 变量的类型为：

```
Int.()->Int
```

该类型就是在普通函数类型的前面添加了一个接收者类型进行限定。

如果接收者类型可通过上下文推断出来，那么 Kotlin 允许使用 Lambda 表达式作为带接收者的匿名函数。例如如下程序。

程序清单：codes\08\8.1\AnnoymousTest.kt

```
class HTML {
    fun body() {
        println("  <body></body>")
    }
    fun head() {
        println("  <head></head>")
    }
}
// 定义一个类型为 HTML.() -> Unit 的形参（带接收者的匿名函数）
// 这样在函数中 HTML 对象就增加了一个 init 方法
fun html(init: HTML.() -> Unit) {
    println("<html>")
    val html = HTML() // 创建接收者对象
    html.init()    // 使用接收者调用 init 引用匿名函数（即传入的参数）
```

```
        println("</html>")
    }
    fun main(args: Array<String>) {
        // 调用 html 函数，需要传入 HTML.() -> Unit 类型的参数
        // 此时系统可推断出接收者的类型，故可用 Lambda 表达式代替匿名函数
        html { // Lambda 表达式中的 this 就是该方法的调用者
            head()
            body()
        }
    }
```

上面程序先定义了一个 HTML 类，并在该类中定义了 body()和 head()两个方法，这个类没有任何特别的地方。

接下来程序定义了一个 html()函数，该函数的形参是 HTML.()->Unit 类型，即带接收者的函数类型。html()函数的形参名为 init，这意味着当程序调用 html()函数时传入的匿名函数（或 Lambda 表达式）会传给该 init 参数，这样在该 html()函数内，HTML 对象就被动态扩展了一个 init()方法，而且该方法的执行体是动态传入的。

在 main()函数中调用 html()函数，调用语法看上去有点奇怪，这一点在 6.7 节有介绍：如果调用函数只有一个 Lambda 表达式参数，则可以省略调用函数的圆括号，将 Lambda 表达式放在函数之外即可。也就是说，main()函数中代码的完整形式如下：

```
html({
    head()
    body()
})
```

理解上面的用法很有益，Kotlin 的一个重要用法示例就是类型安全的 Groovy 风格的构建器，其实就是上面这种用法的大量应用。

8.1.7 何时使用扩展

介绍了这么多关于扩展的知识，可能读者有一个疑问：扩展无非就是为类增加方法或属性，为什么不直接在类中定义这些方法和属性，还要通过扩展来定义呢？这不是多此一举吗？

对于一个一直使用 Java 编程的程序员而言，要理解这个问题稍稍有点费力；但对于一个有 C#、Swift 等编经验的读者来说，这个问题的答案是不言而喻的。很多现代编程语言都已支持扩展，由此可见扩展的魅力。



提示：

我在面试一些有经验的 Java 开发者时，往往喜欢问：你觉得 Java 有什么缺点？问这个问题并不代表我不喜欢 Java，更不代表 Java 可能被淘汰。相反，这代表了我对 Java 的喜爱：知其雄，守其雌，为天下谿。了解 Java 的强大和广泛应用，但也清醒地了解 Java 所存在的不足，这样才算真正的喜欢。Java 缺乏扩展机制就是一个不灵活的地方。

扩展的作用主要有如下两个方面。

- 扩展可动态地为已有的类添加方法或属性。
- 扩展能以更好的形式组织一些工具方法。

关于上面第一点，有时候我们要为一些已有框架或库的类增加额外的功能，如果使用 Java，

则只能通过派生子类来实现，但一方面派生子类有一些限制，另一方面派生子类的性能开销也比较大（关于这一点请参考《疯狂 Java 讲义》中“继承与组合”一节），而扩展的出现很好地解决了这个问题。

扩展是一种非常灵活的动态机制，它既不需要使用继承，也不需要使用类似于装饰者的任何设计模式，即可为现有的类增加功能，因此使用非常方便。

关于上面第二点，我们知道 Java 系统提供了 Arrays、Collections、Files 等各种类，还有第三方库提供的大量的 StringUtils、BeanUtils 等类，这些类的作用非常明确：工具类，包含了操作特定类型的工具方法。比如 Arrays 包含了大量操作数组的工具方法，Collections 包含了大量操作集合的工具方法。例如要使用 Collections 对 List 集合元素排序，则需要使用如下代码：

```
Collections.sort(list)
```

上面这行代码平白无故地多出一个 Collections 工具类，而且这也不符合面向对象的代码风格（面向对象的代码风格是：猪八戒.吃(西瓜)），这里多出来的 Collections 类真让人尴尬。

其实希望使用如下更简洁的代码：

```
list.sort()
```

此时就需要让 List 集合本身具有 Collections 类提供的工具方法，通过扩展即可为 List 集合增加这些工具方法。

8.2 final 和 open 修饰符

final 关键字可用于修饰类、属性和方法，表示它修饰的类、属性和方法不可改变。

Kotlin 有一个非常特别的设计：它会为非抽象类自动添加 final 修饰符，也会为非抽象方法、非抽象属性等无须重写的成员自动添加 final 修饰符。如果开发者希望取消 Kotlin 自动添加 final 修饰符，则可使用 open 修饰符，open 修饰符与 final 修饰符是反义词。

此外，Kotlin 与 Java 的一个重大区别是：Kotlin 的 final 修饰符不能修饰局部变量，因此 open 自然也不能修饰局部变量。



Kotlin 的 final 不能修饰局部变量。



8.2.1 可执行“宏替换”的常量

在《疯狂 Java 讲义》中介绍过，使用 final 修饰“宏变量”，该“宏变量”在编译阶段就会被替换掉。

但 Kotlin 不允许使用 final 修饰局部变量，也不允许直接在 Kotlin 类中定义成员变量（Kotlin 定义的是属性），因此 Kotlin 不可能用 final 定义“宏变量”。

Kotlin 提供了 const 用来修饰可执行“宏替换”的常量，这种常量也被称为“编译时”常量，因为它在编译阶段就会被替换掉。

“宏替换”的常量除使用 const 修饰之外，还必须满足如下条件。

- 位于顶层或者是对象表达式的成员。

- 初始值为基本类型值（Java 的 8 种基本类型）或字符串字面值。
- 没有自定义的 getter 方法。

程序清单：codes\08\8.2\ConstValTest.kt

```
// 定义支持“宏替换”的常量
const val MAX_AGE = 100
fun main(args: Array<String>) {
    println(MAX_AGE)
}
```

上面程序中的粗体字代码使用 const 定义了一个支持“宏替换”的常量，并在定义该常量时指定初始值为 100。对于这个程序来说，常量 MAX_AGE 其实根本不存在，当程序执行 println(MAX_AGE)代码时，实际替换为执行 println(100)——而且这个替换在编译阶段就完成了，因此程序在运行阶段完全没有 MAX_AGE 常量。

此外，如果被赋值的表达式只是基本的算术表达式或进行字符串连接运算，没有访问普通变量、常量，调用方法，那么 Kotlin 编译器同样会将这种 const 常量当成“宏变量”处理。示例如下。

程序清单：codes\08\8.2\ConstReplaceTest.kt

```
// 下面定义了 4 个“宏变量”
const val a = 5 + 2
const val b: Double = 1.2 / 3
const val str: String = "疯狂" + "Java"
const val book: String = "疯狂 Java 讲义：" + 99.0
// 下面的 book2 常量的值因为使用了另一个常量，所以无法在编译时被确定下来
val d = 99.0
const val book2: String = "疯狂 Java 讲义：" + d  //①
fun main(args: Array<String>) {
    println(book === "疯狂 Java 讲义：99.0")
}
```

上面程序中粗体字代码定义了 4 个 const 常量，程序为这 4 个常量指定的初始值要么是算术表达式，要么是进行字符串连接运算。即使字符串连接运算中包含隐式类型转换（将数值转换为字符串），编译器也依然可以在编译时就确定 a、b、str、book 这 4 个常量的值，因此它们都是“宏变量”。

从表面上看，①号代码定义的 book2 与 book 没有太大的区别，只是定义 book2 常量的初始值时使用了另一个常量，因此编译器无法在编译时确定 book2 的值，book2 不会被当成“宏变量”处理。

程序最后一行代码分别判断 book 和“疯狂 Java 讲义：99.0”是否相等。由于 book 是一个“宏变量”，它将被直接替换成“疯狂 Java 讲义：99.0”，因此 book 和“疯狂 Java 讲义：99.0”相等。

8.2.2 final 属性

final 属性表明该属性不能被重写，而且如果程序对属性不使用任何修饰符，Kotlin 会自动为该属性添加 final 修饰。例如如下代码。

程序清单：codes\08\8.2\FinalPropertyTest.kt

```
open class FinalPropertyTest{
```

```
    // Kotlin 自动为该属性添加 final 修饰
    var test: String = "测试属性"
}
class Sub: FinalPropertyTest(){
    // 下面的属性定义将出现编译错误，不能重写 final 属性
    override var test: String = "子类属性"
}
```

从上面程序中可以看出，如果一个属性没有使用 open 修饰，Kotlin 会自动为该属性添加 final 修饰，那么子类就不可以重写该属性。

8.2.3 final 方法

使用 final 修饰的方法不可被重写。与属性设计类似的是，如果程序不为方法添加任何修饰符，Kotlin 会自动为该方法添加 final 修饰。

下面程序试图重写 final 方法，将会引发编译错误。

程序清单：codes\08\8.2\FinalMethodTest.kt

```
open class FinalMethodTest{
    fun test(){}
}
class Sub: FinalMethodTest(){
    // 下面的方法定义将出现编译错误，不能重写 final 方法
    override fun test(){}
}
```

上面程序中父类是 FinalMethodTest，该类中定义的 test()方法没有使用 open 修饰，Kotlin 会自动为它添加 final 修饰，因此它是一个 final 方法，如果其子类试图重写该方法，将会引发编译错误。

对于一个 private 方法、属性，因为它仅在当前类中可见，其子类无法访问它，所以子类无法重写该方法、属性——如果在子类中定义一个与父类的 private 方法有相同方法名、相同形参列表、相同返回值类型的方法，则不是方法重写，只是重新定义了一个新方法。因此，即使使用 final 修饰一个 private 访问权限的方法，也依然可以在其子类中定义与该方法具有相同方法名、相同形参列表、相同返回值类型的方法。

下面程序示范了如何在子类中“重写”父类的 private final 方法、属性。

程序清单：codes\08\8.2\PrivateFinalMethodTest.kt

```
open class PrivateFinalMemberTest{
    // 其实下面的方法、属性使用 private 修饰是多余的
    private final fun test(){}
    private final var name: String = "父类属性"
}
class Sub: PrivateFinalMemberTest(){
    // 下面的方法定义不会出现问题
    public fun test(){}
    public var name: String = "子类属性"
}
```

上面程序没有任何问题，虽然子类和父类都包含了同名的 test()->Unit 方法，但子类并没有重写父类的方法，因此即使父类的 test()->Unit 方法使用了 final 修饰，在子类中也依然可以定义 test()->Unit 方法。上面程序的父类中定义的 name 属性也可被子类重写。

使用 final 修饰的方法仅仅是不能被重写，并不是不能被重载，因此下面程序完全没有问题。

```
class FinalOverload{
    // 使用 final 修饰的方法只是不能被重写，但完全可以被重载
    final fun test(){}
    final fun test(arg: String){}
}
```

8.2.4 final 类

使用 final 修饰的类不可以有子类，与方法、属性的设计相同：如果一个类没有显式使用 open 修饰符修饰，那么 Kotlin 会自动为该类添加 final 修饰。

当子类继承父类时，将可以访问到父类内部数据，并可通过重写父类方法来改变其实现细节，这可能会导致一些不安全的因素。Kotlin 出于更严格的设计哲学，为所有类自动添加 final 修饰，如果希望某个类可以派生子类，则需要为该类添加 open 修饰。

下面程序示范了没有 open 修饰符的类不能派生子类。

```
class FinalClass {}
// 下面的类定义将出现编译错误
class Sub: FinalClass() {}
```

因为 FinalClass 类没有使用 open 修饰，因此它是一个 final 类，而 Sub 试图继承 FinalClass 类，这将会引起编译错误。

8.2.5 不可变类

不可变（immutable）类的意思是创建该类的实例后，该实例的属性值是不可改变的。

如果需要创建自定义的不可变类，可遵守如下规则。

- 提供带参数的构造器，用于根据传入的参数来初始化类中的属性。
- 定义使用 final 修饰的只读属性，避免程序通过 setter 方法改变该属性值。

如果有必要，则重写 Any 类的 hashCode()和 equals()方法。equals()方法将关键属性作为两个对象是否相等的标准。除此之外，还应该保证两个用 equals()方法判断为相等的对象的 hashCode()也相等。

例如，String 这个类就做得很好，它就是将 String 对象中的字符序列作为相等的标准，其 hashCode()方法也是根据字符序列计算得到的。

下面定义一个不可变的 Address 类，程序把 Address 类的 detail 和 postCode 属性都定义成只读属性，并使用 final 修饰这两个属性，不允许其他方法修改这两个属性的值。

程序清单：codes\08\8.2\Address.kt

```
// 定义可初始化两个属性的构造器
class Address(val detail:String, val postCode: String){
    // 重写 equals()方法，判断两个对象是否相等
    override operator fun equals(other: Any?): Boolean{
        if (this == other){
            return true
        }
        if(other == null){
            return false
        }
```

```
        if(other.javaClass == Address::class){
            var ad = other as Address
            // 当 detail 和 postCode 相等时，可认为两个 Address 对象相等
            return this.detail.equals(ad.detail)
                && this.postCode.equals(ad.postCode)
        }
        return false
    }
    override fun hashCode(): Int{
        return detail.hashCode() + postCode.hashCode() * 31
    }
}
```

对于上面的 Address 类，当程序创建了 Address 对象后，同样无法修改该 Address 对象的 detail 和 postCode 属性。

与不可变类对应的是可变类，其含义是该类的对象的属性值是可变的。大部分时候所创建的类都是可变类，只要我们定义了任何读写属性，该类就是可变类。

与可变类相比，不可变类的实例在整个生命周期中永远处于初始化状态，它的属性值不可改变。因此，对不可变类的对象的控制将更加简单。

有一个问题需要说明：当创建不可变类时，如果它包含的成员属性的类型是可变的，那么其对象的属性值依然是可改变的——这个不可变类其实是失败的。

下面程序试图定义一个不可变的 Person 类，但因为 Person 类包含一个可变类型的属性，所以导致 Person 类也变成了可变类。

程序清单：codes\08\8.2\Person.kt

```
class Name(var firstName: String = "", var lastName: String = ""){
}
class Person(val name: Name){
}
fun main(args: Array<String>){
    val n = Name("悟空", "孙")
    var p = Person(n)
    // Person 对象的 name 的 firstName 值为"悟空"
    println(p.name.firstName)
    // 改变 Person 对象的 name 的 firstName 值
    n.firstName = "八戒"
    // Person 对象的 name 的 firstName 值被改为"八戒"
    println(p.name.firstName)
}
```

上面程序中粗体字代码修改了 Name 对象（可变类的实例）的 firstName 的值，但由于 Person 类的 name 属性引用了该 Name 对象，就会导致 Person 对象的 name 的 firstName 的值会被改变，这就破坏了设计 Person 类的初衷。

为了保持 Person 对象的不可变性，必须保护好 Person 对象的引用类型的属性：name，让程序无法访问到 Person 对象的 name 属性的幕后变量，也就无法利用 name 属性的可变性来改变 Person 对象了。为此，将 Person 类改为如下形式：

```
class Person{
    val name: Name
        // 返回一个新的对象，该对象的 firstName 和 lastName
        // 与该 Person 对象里的幕后字段的 firstName 和 lastName 相同
        get() = Name(field.firstName, field.lastName)
```

```
    constructor(name: Name){
        // 设置 name 属性值为新创建的 Name 对象，该对象的 firstName 和 lastName
        // 与传入的 name 参数的 firstName 和 lastName 相同
        this.name = Name(name.firstName, name.lastName)
    }
}
```

注意阅读上面程序中的粗体字代码，Person 类改写了设置 name 属性的方法，也改写了 name 属性的 getter 方法。当程序向 Person 构造器里传入一个 Name 对象时，该构造器创建 Person 对象时并不是直接利用已有的 Name 对象（利用已有的 Name 对象有风险，因为这个已有的 Name 对象是可变的，如果程序改变了这个 Name 对象，将会导致 Person 对象也发生变化），而是重新创建了一个 Name 对象来赋给 Person 对象的 name 属性。当 Person 对象返回 name 属性时，它并没有直接返回 name 属性的幕后字段，因为直接返回 name 属性的幕后字也可能导致它所引用的 Name 对象被修改。

如果将 Person 类定义改为上面形式，再次运行 codes\08\8.2\Person.kt 程序，将看到 Person 对象的 name 的 firstName 不会被修改。

因此，如果需要设计一个不可变类，尤其要注意其引用类型的属性，如果属性的类型本身是可变的，就必须采取必要的措施来保护该属性所引用的对象不会被修改，这样才能创建真正的不可变类。

8.3 抽象类

当编写一个类时，常常会为该类定义一些方法，这些方法用以描述该类的行为方式，这些方法都有具体的方法体。但在某些情况下，某个父类只是知道其子类应该包含怎样的方法，但无法准确地知道这些子类如何实现方法。例如定义一个 Shape 类，这个类应该提供一个计算周长的方法 calPerimeter()，但不同的 Shape 子类对周长的计算方法是不一样的，即 Shape 类无法准确地知道其子类计算周长的方法。

可能有读者会提出，既然 Shape 类不知道如何实现 calPerimeter()方法，那就干脆不要管它了！这不是一个好思路：假设有一个 Shape 变量，该变量实际上引用到 Shape 子类的实例，那么这个 Shape 变量就无法调用 calPerimeter()方法（必须将其强制转换为其子类类型，才可调用 calPerimeter()方法），这就降低了程序的灵活性。

那么如何既能让 Shape 类中包含 calPerimeter()方法，又无须提供其方法实现呢？使用抽象方法即可满足该要求：抽象方法是只有方法签名，没有方法实现的方法。

需要说明的是，有 abstract 修饰的成员，无须使用 open 修饰，当使用 abstract 修饰类时，表明这个类需要被继承；当使用 abstract 修饰方法、属性时，表明这个方法、属性必须由子类提供实现（即重写）。而使用 final 修饰的类不能被继承，使用 final 修饰的方法不能被重写。因此 final 和 abstract 永远不能同时使用。

但是，抽象类中的具体方法、属性依然有 final 修饰，如果程序需要重写抽象类中的具体方法、属性，则依然需要显式为这些方法、属性添加 open 修饰。

8.3.1 抽象成员和抽象类

抽象成员（方法和属性）和抽象类必须使用 abstract 修饰符来定义，包含抽象成员的类只能被定义成抽象类，抽象类中可以没有抽象成员。

抽象方法和抽象类的规则如下。

- 抽象类必须使用 abstract 修饰符来修饰，抽象成员也必须使用 abstract 修饰符来修饰，抽象方法不能有方法体。
- 抽象类不能被实例化，无法调用抽象类的构造器创建抽象类的实例。即使抽象类中不包含任何抽象成员，这个抽象类也不能创建实例。
- 抽象类可以包含属性、方法（普通方法和抽象方法都可以）、构造器、初始化块、嵌套类（接口、枚举）5 种成员。抽象类的构造器不能用于创建实例，主要用于被其子类调用。
- 含有抽象成员的类（包括直接定义了一个抽象成员；或继承了一个抽象父类，但没有完全实现父类包含的抽象成员；或实现了一个接口，但没有完全实现接口包含的抽象成员三种情况）只能被定义成抽象类。

注意：

归纳起来，抽象类可用“有得有失”4 个字来描述。“得”指的是抽象类多了一个能力：抽象类可以包含抽象成员；“失”指的是抽象类失去了一个能力：抽象类不能用于创建实例。



定义抽象方法，只需在普通方法上增加 abstract 修饰符，并把普通方法的方法体（也就是方法后用花括号括起来的部分）全部去掉即可。

注意：

抽象方法和空方法体的方法不是同一个概念。例如，fun test()是一个抽象方法，它根本没有方法体，即方法定义后面没有一对花括号；但 fun test(){}是一个普通方法，它已经定义了方法体，只是方法体为空，即它的方法体什么也不做，因此这个方法不可使用 abstract 来修饰。



定义抽象类，只需在普通类上增加 abstract 修饰符即可。甚至一个普通类（没有包含抽象方法的类）增加 abstract 修饰符后也将变成抽象类。

下面定义一个 Shape 抽象类。

程序清单：codes\08\8.3\Shape.kt

```
abstract class Shape{
    init{
        println("执行 Shape 的初始化块...")
    }
    var color = ""
    // 定义一个计算周长的抽象方法
    abstract fun calPerimeter(): Double
    // 定义一个代表形状的抽象的只读属性
    // 抽象属性不需要初始值
    abstract val type: String
    // 定义 Shape 的构造器，该构造器并不是用于创建 Shape 对象的
    // 而是用于被子类调用
    constructor(){}
    constructor(color: String){
```

```
        println("执行 Shape 的构造器...")
        this.color = color
    }
}
```

上面的Shape类中包含了一个抽象方法calPerimeter()和一个抽象属性type，所以这个Shape类只能被定义成抽象类。Shape类中既包含了初始化块，也包含了构造器，这些都不是在创建Shape对象时被调用的，而是在创建其子类的实例时被调用。

注意：

抽象类不能用于创建实例，只能当作父类被其子类继承。



下面定义一个三角形类，这个三角形类被定义成普通类，因此必须实现Shape类中的所有抽象成员（包括抽象方法和抽象属性）。

程序清单：codes\08\8.3\Triangle.kt

```
// 定义三角形的三边
class Triangle(color: String, var a: Double,
    var b: Double, var c: Double): Shape(color){
    fun setSides(a: Double, b: Double, c: Double){
        if (a >= b + c || b >= a + c || c >= a + b){
            println("三角形两边之和必须大于第三边")
            return
        }
        this.a = a
        this.b = b
        this.c = c
    }
    // 重写 Shape 类的计算周长的抽象方法
    override fun calPerimeter(): Double{
        return a + b + c
    }
    // 重写 Shape 类的代表形状的抽象属性
    override val type: String = "三角形"
}
```

上面的Triangle类继承了Shape抽象类，并实现了Shape类中的抽象方法和抽象属性，因此是一个普通类，故可以创建Triangle类的实例，也可以让一个Shape类型的变量引用Triangle对象。

下面再定义一个Circle普通类，Circle类也是Shape类的一个子类。

程序清单：codes\08\8.3\Circle.kt

```
class Circle(color: String, var radius: Double): Shape(color){
    // 重写 Shape 类的计算周长的抽象方法
    override fun calPerimeter(): Double = 2 * Math.PI * radius
    // 重写 Shape 类的代表形状的抽象属性
    override val type: String = "圆形"
}
fun main(args: Array<String>){
    var s1: Shape = Triangle("黑色" , 3.0 , 4.0, 5.0)
    var s2: Shape = Circle("黄色" , 3.0)
    println(s1.type)
    println(s1.calPerimeter())
```

```
    println(s2.type)
    println(s2.calPerimeter())
}
```

上面main()函数中定义了两个Shape类型的变量，它们分别指向Triangle对象和Circle对象。由于在Shape类中定义了calPerimeter()方法和type属性，所以程序可以直接调用s1和s2变量的calPerimeter()方法和type属性，无须强制类型转换为其子类类型。

利用抽象类和抽象方法的优势，可以更好地发挥多态的优势，使得程序更加灵活。

注意：

abstract不能用于修饰局部变量，Kotlin没有抽象变量的说法；abstract也不能用于修饰构造器，没有抽象构造器，抽象类中定义的构造器只能是普通构造器。



注意：

使用abstract关键字修饰的方法必须被其子类重写才有意义，否则这个方法将永远不会有方法体，因此abstract方法不能定义为private访问权限，即private和abstract不能同时修饰方法。



与Java类似的是，Kotlin也允许使用抽象成员重写非抽象成员。例如如下代码：

```
open class Base {
    open fun foo() {}
}
abstract class Sub : Base() {
    override abstract fun foo()
}
```

上面程序中抽象的foo()方法重写了父类的非抽象的foo()方法。

8.3.2 抽象类的作用

从前面的示例程序可以看出，抽象类不能创建实例，只能当成父类来被继承。从语义的角度来看，抽象类是从多个具体类中抽象出来的父类，它具有更高层次的抽象。从多个具有相同特征的类中抽象出一个抽象类，以这个抽象类作为其子类的模板，从而避免了子类设计的随意性。

抽象类体现的就是一种模板模式的设计，抽象类作为多个子类的通用模板，子类在抽象类的基础上进行扩展、改造，但子类总体上会大致保留抽象类的行为方式。

如果编写一个抽象父类，父类提供了多个子类的通用方法，并把一个或多个方法留给其子类实现，这就是一种模板模式，模板模式也是十分常见且简单的设计模式之一。例如前面介绍的Shape、Circle和Triangle三个类，已经使用了模板模式。下面再介绍一个模板模式的范例，在这个范例的抽象父类中，父类的普通方法依赖于一个抽象方法，而抽象方法则推迟到子类中提供实现。

程序清单：codes\08\8.3\SpeedMeter.kt

```
// 定义带转速属性的主构造器
abstract class SpeedMeter(var turnRate: Double){
    // 把返回车轮半径的方法定义成抽象方法
```

```
    abstract fun calGirth(): Double
    // 定义计算速度的通用算法
    fun getSpeed(): Double{
        // 速度等于 车轮周长 * 转速
        return calGirth() * turnRate
    }
}
```

上面程序定义了一个抽象的 SpeedMeter 类(车速表),该类中定义了一个 getSpeed()方法,该方法用于返回当前车速,getSpeed()方法依赖于 calGirth()方法的返回值。对于一个抽象的 SpeedMeter 类而言,它无法确定车轮的周长,因此 calGirth()方法必须推迟到其子类中实现。

下面是其子类 CarSpeedMeter 的代码,该类实现了其抽象父类的 calGirth()方法,既可创建 CarSpeedMeter 类的对象,也可通过该对象来取得当前速度。

程序清单:codes\08\8.3\CarSpeedMeter.kt

```
public class CarSpeedMeter(var radius: Double): SpeedMeter(0.0){
    override fun calGirth(): Double{
        return radius * 2 * Math.PI
    }
}
fun main(args: Array<String>) {
    val csm = CarSpeedMeter(0.28)
    csm.turnRate = 15.0
    println(csm.getSpeed())
}
```

SpeedMeter 类中提供了车速表的通用算法,但一些具体的实现细节则推迟到其子类 CarSpeedMeter 中实现。这也是一种典型的模板模式。

模板模式在面向对象的软件中很常用,其原理简单,实现也很简单。下面是使用模板模式的一些简单规则。

- 抽象父类可以只定义需要使用的某些方法,把不能实现的部分抽象成抽象方法,留给其子类去实现。
- 父类中可能包含需要调用其他系列方法的方法,这些被调方法既可以由父类实现,也可以由其子类实现。父类中提供的方法只是定义了一个通用算法,其实现也许并不完全由自身来完成,而必须依赖于其子类的辅助。

8.3.3 密封类

密封类这个名字很奇怪,光看名字很容易把它当成被“密封”的类,以为它不能派生子类。但实际上,密封类是一种特殊的抽象类,专门用于派生子类。

密封类与普通抽象类的区别在于:密封类的子类是固定的。密封类的子类必须与密封类本身在同一个文件中,在其他文件中则不能为密封类派生子类,这样就限制了在其他文件中派生子类。



提示:

在 Kotlin 1.1 之前,密封类的子类必须在密封类的内部声明。

例如，如下程序定义了一个密封类和两个子类。

程序清单：codes\08\8.3\SealedTest.kt

```
// 定义一个密封类，其实就是抽象类
sealed class Apple{
    abstract fun taste()
}
open class RedFuji: Apple(){
    override fun taste(){
        println("红富士苹果香甜可口")
    }
}
data class Gala(var weight: Double): Apple(){
    override fun taste(){
        println("嘎拉果更清脆，重量为：${weight}")
    }
}
fun main(args: Array<String>){
    // 使用 Apple 声明变量，用子类实例赋值
    var ap1: Apple = RedFuji()
    var ap2: Apple = Gala(2.3)
    ap1.taste()
    ap2.taste()
}
```

上面程序中定义了一个密封类，接下来即可在该密封类中定义抽象方法，由此可见，密封类的本质就是抽象类。

注意：

密封类经过 Kotlin 编译器编译之后就得到一个抽象类的 class 文件，只不过该抽象类的普通构造器都会声明为 private 权限，而 Kotlin 会为之创建一个对应的带 kotlin.jvm.internal.DefaultConstructorMarker 参数的构造器。



定义密封类之后，就像定义了抽象类，接下来即可在同一个文件中为该密封类派生子类，就像为普通抽象类派生子类一样，如上面程序代码所示。

根据密封类的规则我们知道，密封类的所有构造器都必须是 private 的，不管开发者是否使用 private 修饰，系统都会为之自动添加 private 修饰。

最后有一点需要说明的是，密封类的直接子类必须与密封类位于同一个文件中，但密封类的间接子类（子类的子类）则无须在同一个文件中。例如，我们可以为上面的 RedFuji 派生如下子类：

```
class JapaneseFuji: RedFuji(){
}
```

上面代码定义的 JapaneseFuji 类则无须位于 SealedTest.kt 文件中。

使用密封类的好处是：它的子类是固定的，编译器可以清楚地知道密封类只可能有固定数量的子类。因此使用 when 表达式判定密封类时，编译器可以清楚地知道是否覆盖了所有情况，从而判断是否需要添加 else 子句。例如如下代码：

```
fun judge(ap: Apple){
    when (ap) {
        is RedFuji -> println("红富士苹果")
```

```
        is Gala -> println("嘎拉果")
    }
}
```

8.4 接口

Kotlin 的接口是以 Java 8 接口为蓝本设计的，因此 Kotlin 的接口与 Java 8 的接口非常相似。

8.4.1 接口的定义

和类定义不同，定义接口时不再使用 class 关键字，而是使用 interface 关键字。接口定义的基本语法如下：

```
[修饰符] interface 接口名: 父接口 1, 父接口 2...{
    零个到多个属性定义...
    零个到多个方法定义...
    零个到多个嵌套类、嵌套接口、嵌套枚举定义...
}
```

对上面语法的详细说明如下。

- 修饰符可以是 public | internal | private 中的任意一个，或完全省略修饰符。如果省略了访问控制符，则默认采用 public。
- 接口名应与类名采用相同的命名规则，即如果仅从语法的角度来看，接口名只要是合法的标识符即可；如果要遵守 Kotlin 的可读性规范，则接口名应由多个有意义的单词连缀而成，每个单词首字母大写，单词与单词之间无须任何分隔符。接口名通常使用形容词。
- 一个接口可以有多个直接父接口，但接口只能继承接口，不能继承类。

与 Java 8 相似的是，Kotlin 的接口既可包含抽象方法，也可包含非抽象方法，这是非常自由的。但接口中的属性没有幕后字段，因此无法保存状态，所以接口中的属性要么声明为抽象属性，要么为之提供 setter、getter 方法。



提示：

关于上面语法中的嵌套类、嵌套接口、嵌套枚举都属于嵌套类型，下一节将会详细介绍嵌套类的相关知识。

由于接口定义的是一种规范，因此接口中不能包含构造器和初始化块定义。对比接口和类的定义方式，不难发现接口中的成员比类中的成员少了两种。

接口中定义的方法既可是抽象方法，也可是非抽象方法。如果一个方法没有方法体，Kotlin 会自动为该方法添加 abstract 修饰符；同理，如果一个只读属性没有定义 getter 方法，Kotlin 会自动为该属性添加 abstract 修饰符；如果一个读写属性没有定义 getter、setter 方法，Kotlin 会自动为该属性添加 abstract 修饰符。

Kotlin 接口与 Java 接口还有一点区别：Java 接口中的所有成员都会自动使用 public 修饰，如果为这些成员指定访问权限，也只能指定 public 访问权限；但 Kotlin 接口中的成员可支持 private 和 public 两种访问权限。具体规则如下。

- 对于需要被实现类重写的成员，如抽象方法、抽象属性，只能使用 public 修饰。如果要添加访问控制符，则只能用 public；如果不加访问权限，则系统默认添加 public。

- 对于不需要被实现类重写的成员，如非抽象方法、非抽象属性、嵌套类（包括嵌套抽象类）、嵌套接口、嵌套枚举，都可使用 private 或 public 修饰，我们可以根据需要添加 private 修饰符将这些成员限制在接口内访问；如果不加访问权限，则系统默认添加 public。

下面定义一个接口。

程序清单：codes\08\8.4\Outputable.kt

```
interface Outputable{
    // 只读属性定义了 getter 方法，非抽象属性
    val name: String
        get() = "输出设备"
    // 只读属性没有定义 getter 方法，抽象属性
    val brand: String
    // 读写属性没有定义 getter、setter 方法，抽象属性
    var category: String
    // 接口中定义的抽象方法
    fun out()
    fun getData(msg: String)
    // 在接口中定义的非抽象方法，可使用 private 修饰
    fun print(vararg msgs: String){
        for (msg in msgs){
            println(msg)
        }
    }
    // 在接口中定义的非抽象方法，可使用 private 修饰
    fun test(){
        println("接口中 test()方法")
    }
}
```

上面定义了一个 Outputable 接口，这个接口还定义了两个抽象方法：表示取得数据的 getData()方法和表示输出的 out()方法。这就定义了 Outputable 接口的规范：只要某个类能取得数据，并可以将数据输出，那么它就是一个输出设备。至于这个设备的实现细节，这里暂时不用关心。

Kotlin 允许在接口中定义普通方法，上面的 Outputable 接口中定义了 print()和 test()两个非抽象方法，它们都有方法体，因此系统不会自动为它们添加 abstract 修饰符。由于这两个方法是非抽象方法，因此我们可以为它们添加 private 修饰符，如果不添加任何修饰符，则系统默认为它们添加 public 修饰符。

上面接口中还定义了三个属性，其中 name 是一个只读属性，程序为该属性定义了 getter 方法，因此该属性不是抽象属性，系统无须为之添加 abstract 修饰符；brand 是只读属性，没有 getter 方法，它是抽象属性，系统会自动为之添加 abstract 修饰符；category 是读写属性，程序没有为之定义 getter 和 setter 方法，它也是抽象属性，系统会自动为之添加 abstract 修饰符。

8.4.2 接口的继承

接口的继承和类继承不一样，接口完全支持多继承，即一个接口可以有多个直接父接口。和类继承相似，子接口继承某个父接口，将会获得父接口中定义的所有方法、属性。

一个接口继承多个父接口时，多个父接口排在英文冒号（:）之后，它们之间以英文逗号

(,) 隔开。下面程序定义了三个接口，其中第三个接口继承了前两个接口。

程序清单：codes\08\8.4\InterfaceExtendsTest.kt

```
interface InterfaceA{
    val propA: Int
        get() = 5
    fun testA()
}
interface InterfaceB{
    val propB: Int
        get() = 6
    fun testB()
}
interface InterfaceC: InterfaceA, InterfaceB{
    val propC: Int
        get() = 7
    fun testC()
}
```

上面程序中的 InterfaceC 接口继承了 InterfaceA 和 InterfaceB 接口，由此可见，接口完全支持多继承。

8.4.3 使用接口

接口不能用于创建实例，但可以用于声明变量。当使用接口来声明变量时，这个引用类型的变量必须引用到其实现类的对象。除此之外，接口的主要用途就是被实现类实现。归纳起来，接口主要有如下用途。

- 定义变量，也可用于进行强制类型转换。
- 被其他类实现。

一个类可以实现一个或多个接口，直接将被实现的多个接口、父类放在英文冒号之后，且父类、接口之间没有顺序要求，只要将它们用英文逗号隔开即可。

一个类可以实现多个接口，这是 Kotlin 为单继承的灵活性不足所做的补充。

```
[修饰符] class 类名: 父类, 接口 1,接口 2... {
    类体部分
}
```

实现接口与继承父类相似，一样可以获得所实现接口中定义的属性（包括抽象属性和非抽象属性）、方法（包括抽象方法和非抽象方法）。

一个类实现了一个或多个接口之后，这个类必须完全实现这些接口中所定义的全部抽象成员（也就是重写这些抽象方法和抽象属性）；否则，该类将保留从父接口那里继承到的抽象成员，该类也必须定义成抽象类。

一个类实现某个接口时，该类将会获得接口中定义的属性、方法等，因此可以把实现接口理解为一种特殊的继承，相当于实现类继承了一个更抽象的类。

下面看一个实现接口的类。

程序清单：codes\08\8.4\Printer.kt

```
// 定义一个 Product 接口
interface Product{
    fun getProduceTime(): Int
}
```

```
const val MAX_CACHE_LINE = 10
// 让 Printer 类实现 Outputable 和 Product 接口
class Printer: Outputable, Product{
    private val printData
        = Array<String>(MAX_CACHE_LINE, {""})
    // 用以记录当前需打印的作业数
    private var dataNum = 0
    // 重写接口的抽象只读属性
    override val brand: String = "HP"
    // 重写接口的抽象读写属性
    override var category: String = "输出外设"
    override fun out(){
        // 只要还有作业，就继续打印
        while(dataNum > 0){
            println("打印机打印：" + printData[0])
            // 把作业队列整体前移一位，并将剩下的作业数减 1
            System.arraycopy(printData , 1
                , printData, 0, --dataNum)
        }
    }
    override fun getData(msg: String){
        if (dataNum >= MAX_CACHE_LINE){
            println("输出队列已满，添加失败")
        } else {
            // 把打印数据添加到队列里，已保存数据的数量加 1
            printData[dataNum++] = msg
        }
    }
    override fun getProduceTime(): Int{
        return 45
    }
}
fun main(args: Array<String>){
    // 创建一个 Printer 对象，当成 Output 使用
    var o: Outputable = Printer()
    o.getData("轻量级 Java EE 企业应用实战")
    o.getData("疯狂 Java 讲义")
    o.out()
    o.getData("疯狂 Android 讲义")
    o.getData("疯狂 Ajax 讲义")
    o.out()
    // 调用 Outputable 接口中定义的非抽象方法
    o.print("孙悟空" , "猪八戒" , "白骨精")
    o.test()
    // 创建一个 Printer 对象，当成 Product 使用
    val p: Product = Printer()
    println(p.getProduceTime())
    // 所有接口类型的引用变量都可直接赋给 Any 类型的变量
    val obj: Any = p
}
```

从上面程序中可以看出，Printer 类实现了 Outputable 和 Product 接口，因此 Printer 对象既可直接赋值给 Outputable 变量，也可直接赋值给 Product 变量。就好像 Printer 类既是 Outputable 类的子类，也是 Product 类的子类，这就是 Kotlin 提供的模拟多继承。

上面程序中 Printer 实现了 Outputable 接口，即可获取 Outputable 接口中定义的 print()和 test()

两个非抽象方法，因此 Printer 实例可以直接调用这两个默认方法。

注意：

在实现接口中的成员时，必须使用 public 访问控制符，因为接口中的成员都是 public 的，而子类（相当于实现类）重写父类方法时访问权限只能更大或者相等，所以实现类实现接口中的成员时只能使用 public 访问权限。



接口不能显式继承任何类，但所有接口类型的变量都可以直接赋给 Any 类型的变量。所以在上面程序中可以把 Product 类型的变量直接赋给 Any 类型的变量，这是利用向上转型来实现的。因为编译器知道任何 Kotlin 对象都必须是 Any 或其子类的实例，Product 类型的对象也不例外（它必须是 Product 接口实现类的对象，该实现类肯定是 Any 的显式或隐式子类）。

8.4.4 接口和抽象类

接口和抽象类有一些相似之处，它们都具有如下特征。

- 接口和抽象类都不能被实例化，它们都位于继承树的顶端，用于被其他类实现和继承。
- 接口和抽象类都可以包含抽象成员，实现接口或继承抽象类的普通子类都必须实现这些抽象成员。

但接口和抽象类之间的差别也很大，这种差别主要体现在二者的设计目的上。下面具体分析二者的差别。

接口作为系统与外界交互的窗口，体现的是一种规范。对于接口的实现者而言，接口规定了实现者必须向外提供哪些服务（以方法、属性的形式来提供）；对于接口的调用者而言，接口规定了调用者可以调用哪些服务，以及如何调用这些服务（就是如何调用方法、访问属性）。当在一个程序中使用接口时，接口是多个模块之间的耦合标准；当在多个应用程序之间使用接口时，接口是多个程序之间的通信标准。

从某种程度上看，接口类似于整个系统的“总纲”，它制定了系统各模块应该遵循的标准，因此一个系统中的接口不应该经常改变。一旦接口被改变，对整个系统甚至其他系统的影响将是辐射式的，导致系统中大部分类都需要改写。

抽象类则不一样，抽象类作为系统中多个子类的共同父类，所体现的是一种模板模式设计。抽象类作为多个子类的抽象父类，可以被当成系统实现过程中的中间产品，这个中间产品已经实现了系统的部分功能（那些已经提供实现的方法），但这个产品依然不能当成最终产品，必须有更进一步的完善，这种完善可能有几种不同方式。

此外，接口和抽象类在用法上也存在如下差别。

- 接口中不包含构造器；但抽象类中可以包含构造器，抽象类中的构造器并不是用于创建对象的，而是让其子类调用这些构造器来完成属于抽象类的初始化操作。
- 接口中不能包含初始化块；但抽象类中则完全可以包含初始化块。
- 一个类最多只能有一个直接父类，包括抽象类；但一个类可以直接实现多个接口，通过实现多个接口可以弥补 Kotlin 单继承的不足。

8.5 嵌套类和内部类

大部分时候，类被定义成一个独立的程序单元。在某些情况下，也会把一个类放在另一个

类的内部定义，这个定义在其他类内部的类就被称为嵌套类（有的地方也叫寄生类），包含嵌套类的类被称为外部类（有的地方也叫宿主类）。

Java 的内部类可分为两种：静态内部类（有 static 修饰）和非静态内部类（无 static 修饰）。而 Kotlin 则完全取消了 static 修饰符。但实际上 Kotlin 也需要有静态内部类和非静态内部类之分，所以 Kotlin 只得为它们换了个“马甲”。

- **嵌套类（相当于静态内部类）**：只要将一个类放在另一个类中定义，这个类就变成了嵌套类，相当于 Java 中有 static 修饰的静态内部类。
- **内部类（非静态内部类）**：使用 inner 修饰的嵌套类叫内部类，相当于 Java 中无 static 修饰的非静态内部类。

嵌套类主要有如下作用。

- 嵌套类提供了更好的封装，可以把嵌套类隐藏在外部类之内，不允许同一个包中的其他类访问该类。假设需要创建 Cow 类，Cow 类需要组合一个 CowLeg 对象，CowLeg 类只有在 Cow 类中才有效，离开了 Cow 类之后没有任何意义。在这种情况下，就可以把 CowLeg 定义成 Cow 的嵌套类，不允许其他类访问 CowLeg。
- 内部类（相当于 Java 的非静态内部类）成员可以直接访问外部类的私有数据，因为内部类被当成其外部类成员，同一个类的成员之间可以互相访问。但外部类不能访问内部类的实现细节，例如内部类的属性。

从语法的角度来看，定义嵌套类（内部类）与定义外部类的语法大致相同，只是嵌套类（内部类）可使用 protected 修饰，用于表示该嵌套类（内部类）可在其外部类的子类中被访问。

定义嵌套类（内部类）非常简单，只要把一个类放在另一个类的类内部嵌套定义即可。此处的“类内部”包括类中的任何位置，甚至在方法中也可以定义嵌套类（方法中定义的嵌套类被称为局部嵌套类）。嵌套类（内部类）定义的语法格式如下：

```
class OuterClass{
    // 此处可以定义嵌套类、内部类
}
```

大部分时候，嵌套类（内部类）都被作为成员嵌套类（内部类）定义，而不是作为局部嵌套类。成员嵌套类（内部类）是一种与属性、方法、构造器和初始化块相似的成员；局部嵌套类则不是类成员。

嵌套类无须使用任何特殊修饰符，它相当于 Java 的静态内部类；内部类需要使用 inner 修饰，它相当于 Java 的非静态内部类。

在前面经常看到同一个 Kotlin 源文件中定义了多个类，在这种情况下，它们不是嵌套类或内部类的关系，依然是两个互相独立的类。例如下面程序：

```
// 下面 A、B 两个空类互相独立，没有谁是谁的嵌套类或内部类
class A
class B
```

上面两个类定义虽然写在同一个源文件中，但它们互相独立，没有谁是谁的嵌套类（内部类）这种关系。嵌套类或内部类一定是放在另一个类的类体部分（也就是类名后的花括号部分）定义的。

因为嵌套类（内部类）属于其外部类的成员，所以可以使用任意访问控制符如 private、internal、protected 和 public 等修饰。



提示：

外部类的上一级程序单元是包，它不在任何类的里面，因此使用 protected 修饰符没有任何意义。使用 private 修饰符则意味着外部类仅能在定义该类的文件中被访问。

8.5.1 内部类

内部类（等同于 Java 的非静态内部类）相当于外部类的实例成员，因此它可以直接访问外部类的所有成员。

下面程序在 Cow 类中定义了一个 CowLeg 内部类，并在 CowLeg 类的方法中直接访问 Cow 的 private 属性。

程序清单：codes\08\8.5\Cow.kt

```
// 通过主构造器为外部类定义属性
class Cow(var weight: Double = 0.0){
    // 定义一个内部类（用 inner 修饰，相当于 Java 的非静态内部类）
    // 通过主构造器为内部类定义属性
    private inner class CowLeg(var length:Double = 0.0, var color: String = ""){
        // 内部类的方法
        fun info(){
            println("当前牛腿颜色是：${color}，高：${length}")
            // 直接访问外部类的 private 修饰的 foo()方法
            foo() // ①
        }
    }
    fun test(){
        val cl = CowLeg(1.12 , "黑白相间")
        cl.info()
    }
    private fun foo(){
        println("Cow 的 foo 方法")
    }
}
fun main(args: Array<String>){
    val cow = Cow(378.9)
    cow.test()
}
```

上面程序中粗体字代码部分是一个普通的类定义，但因为把这个类定义放在了另一个类的内部，且使用了 inner 修饰，所以它就成了一个内部类，可以使用 private 修饰符来修饰这个类。

外部类 Cow 中包含了一个 test()方法，该方法中创建了一个 CowLeg 对象，并调用了该对象的 info()方法。读者不难发现，在外部类中使用内部类时，与平时使用普通类并没有太大的区别。

编译上面程序，将看到在文件所在路径下生成了两个 class 文件，其中一个是 Cow.class，另一个是 Cow$CowLeg.class，前者是外部类 Cow 的 class 文件，后者是内部类 CowLeg 的 class 文件，即成员嵌套类和成员内部类的 class 文件总是这种形式：OuterClass$InnerClass.class。

前面提到过，在内部类中可以直接访问外部类的 private 成员，上面程序中①号粗体字代码就表示在 CowLeg 类的方法内直接访问其外部类的 private 方法。这是因为在内部类对象中，

保存了一个它所寄生的外部类对象的引用（当调用内部类的方法时，必须使用内部类对象作为调用者，内部类实例必须寄生在外部类实例中）。图 8.1 显示了上面程序运行时的内存示意图。

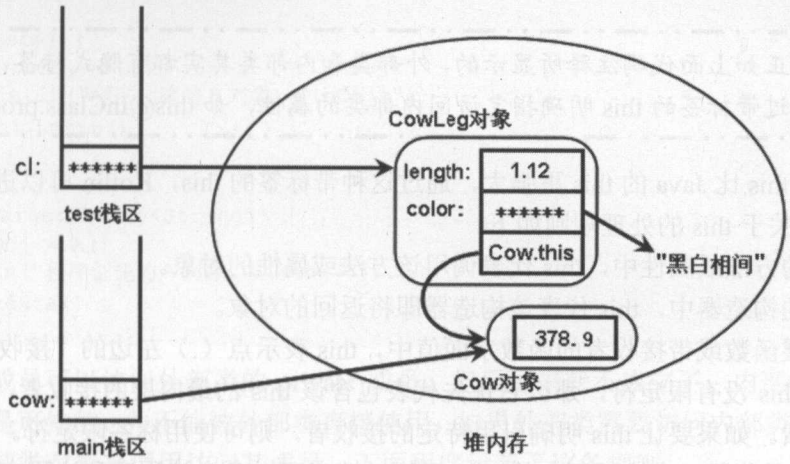


图 8.1 内部类对象中保留外部类对象的引用内存示意图

当在内部类的方法内访问某个属性时，系统优先在该方法内查找是否存在该名字的局部变量，如果存在就使用该变量；如果不存在，则到该方法所在的内部类中查找是否存在该名字的属性，如果存在则使用该属性；如果不存在，则到该内部类所在的外部类中查找是否存在该名字的属性，如果存在则使用该属性；如果依然不存在，系统将出现编译错误，提示找不到该属性。

因此，如果外部类属性、内部类属性与内部类中方法的局部变量同名，则可通过使用 this、带标签的 this 进行限定来区分。例如如下程序。

程序清单：codes\08\8.5\DiscernVariable.kt

```
class DiscernVariable{  // 隐式标签 @DiscernVariable
    private val prop = "外部类的属性"
    inner class InClass {  // 隐式标签 @InClass
        private val prop = "内部类的属性"
        fun info(){
            val prop = "局部变量"
            // 通过外部类类名.this.varName 访问外部类的属性
            println("外部类的属性值：${this@DiscernVariable.prop}")
            // 通过 this.varName 访问内部类的属性
            println("内部类的属性值：${this.prop}")
            // 直接访问局部变量
            println("局部变量的值：${prop}")
        }
    }
    fun test(){
        val ic = InClass()
        ic.info()
    }
}
fun main(args: Array<String>) {
    DiscernVariable().test()
}
```

上面程序中粗体字代码分别访问外部类的属性、内部类的属性。通过带标签的 this 前缀（如

上面第一行粗体字代码所示），可显式指定访问哪个类的属性。不带标签的 this 前缀默认访问内部类的属性。



提示：

正如上面代码注释所显示的，外部类和内部类其实都有隐式标签，因此程序也可通过带标签的 this 明确指定访问内部类的属性，如 this@InClass.prop。

Kotlin 的 this 比 Java 的 this 更强大，通过这种带标签的 this，Kotlin 可以进行非常细致的区分。Kotlin 关于 this 的处理规则如下。

- 在类的方法或属性中，this 代表调用该方法或属性的对象。
- 在类的构造器中，this 代表该构造器即将返回的对象。
- 在扩展函数或带接收者的函数字面值中，this 表示点（.）左边的“接收者”。
- 如果 this 没有限定符，那么它优先代表包含该 this 的最内层的接收者，并且会自动向外搜索。如果要让 this 明确引用特定的接收者，则可使用标签限定符。

如下程序示范了一个比较复杂的 this 示例，读者可以参考该程序和注释来理解 Kotlin 对 this 的处理。

程序清单：codes\08\8.5\ThisTest.kt

```
class A { // 隐式标签 @A
    inner class B { // 隐式标签 @B
        // 为 Int 扩展 foo()方法
        fun Int.foo() { // 隐式标签 @foo
            val a = this@A // A 的 this
            val b = this@B // B 的 this
            val c = this // 不带标签的 this，默认代表该方法所属对象：Int 对象
            val c1 = this@foo // 显式指定@foo 标签，与 c 代表的对象相同
            println(a)
            println(b)
            println(c)
            println(c1)
            // 为 String 扩展 funLit()方法
            val funLit = lambda@ fun String.() {
                val d = this // 不带标签的 this，默认代表该方法所属对象：String 对象
                val d1 = this@lambda // 显式指定@lambda 标签，与 d 代表的对象相同
                println(d)
                println(d1)
            }
            "fkit".funLit()
            // 直接定义一个 Lambda 表达式，没有接收者
            val funLit2 = {
                // 该 this 所在的 Lambda 表达式没有接收者，因此当前范围没有 this
                // 系统会继续向该 Lambda 表达式所在范围搜索 this
                // 故此处 this 将代表 foo()方法的接收者：Int 对象
                val e = this
                val e1 = this@foo // 显式指定@foo 标签，与 e 代表的对象相同
                println("foo()方法中 Lambda 表达式的 this：" + e)
                println("e1 的 this：" + e1)
            }
            funLit2()
        }
        fun testB(){
```

```
            // 调用2（Int值）的foo()方法
            2.foo()
        }
    }
    fun testA(){
        var bObj = B()
        println("程序创建的B对象：${bObj}")
        bObj.testB()
    }
}
fun main(args: Array<String>) {
    var aObj = A()
    println("程序创建的A对象：${aObj}")
    aObj.testA()
}
```

内部类的成员可以访问外部类的 private 成员，但反过来就不成立了。内部类的成员只在内部类范围内是可知的，并不能被外部类直接使用。如果外部类需要访问内部类的成员，则必须显式创建内部类对象来调用访问其成员。下面程序示范了这条规则。

程序清单：codes\08\8.5\Outer.kt

```
class Outer{
    private val outProp = 9
    inner class Inner{
        val inProp = 5
        fun acessOuterProp(){
            // 内部类可以直接访问外部类的private属性
            println("外部类的outProp值:${outProp}")
        }
    }
    fun accessInnerProp(){
        // 外部类不能直接访问内部类的属性
        // 下面代码出现编译错误
//        println("内部类的inProp值:${inProp}")
        // 如需访问内部类的属性，必须显式创建内部类对象
        println("内部类的inProp值: ${Inner().inProp}")
    }
}
fun main(args: Array<String>) {
    // 执行下面代码，只创建了外部类对象，还未创建内部类对象
    val ot = Outer()  // ①
    ot.accessInnerProp()
}
```

程序中粗体字代码试图在外部类的方法中访问内部类的属性，这将引起编译错误。

外部类不允许访问内部类的成员（属性和方法）还有一个原因，上面程序中 main()函数的①号粗体字代码创建了一个外部类对象，并调用外部类对象的 accessInnerProp()方法。此时内部类对象根本不存在，如果允许 accessInnerProp()方法访问内部类对象，将肯定引起错误。

8.5.2 嵌套类

嵌套类相当于 Java 的静态内部类，因此嵌套类直接属于外部类的类本身，而不是外部类实例相关。

Java 语法有一条规则：静态成员不可访问非静态成员。而 Kotlin 彻底取消了 static 修饰符，

因此 Kotlin 类中的成员除嵌套类之外，全部都是非静态成员，因此嵌套类不可访问外部类的其他任何成员（只能访问其他嵌套类）。



注意：

嵌套类不能访问外部类的其他成员，只能访问另一个嵌套类。



下面程序就演示了这条规则。

程序清单：codes\08\8.5\NestedClassTest.kt

```
class NestedClassTest{
    var prop1 = 5
    fun test(){
        println("外部类的 test()方法")
    }
    // 没有 inner 修饰符，是嵌套类（相当于 Java 的静态内部类）
    class NestedClass{
        fun accessOuterMember(){
            // 访问另一个嵌套类是允许的
            val a = A()
            // 下面两行代码都会出现错误
            println(prop1)
            test()
        }
    }
    class A
}
```

上面程序中两行粗体字代码定义了一个属性和一个方法，NestedClass 类中定义了一个accessOuterMember()方法，但它不能访问外部类的 prop1 属性和 test()方法。

嵌套类唯一可访问的是外部类的其他嵌套类。

嵌套类相当于外部类的静态成员，因此外部类的所有方法、属性、初始化块都可以使用嵌套类来定义变量、创建对象等。

外部类依然不能直接访问嵌套类的成员，但可以使用嵌套类的对象作为调用者来访问嵌套类的成员。下面程序示范了这条规则。

程序清单：codes\08\8.5\AccessNestedClass.kt

```
class AccessNestedClass{
    class NestedClass{
        var prop = 9
    }
    fun accessNestedProp(){
        // println(prop)
        // 上面代码出现错误，应改为如下形式
        // 通过对象访问嵌套类的成员
        println(NestedClass().prop)
    }
}
```

除此之外，Kotlin 还允许在接口中定义嵌套类，但不允许在接口中定义内部类（即不允许定义使用 inner 修饰的内部类）。如果为接口中的嵌套类指定访问控制符，则只能指定 public 或 private；如果定义接口中的嵌套类时省略访问控制符，则该嵌套类默认是 public 访问权限。

8.5.3 在外部类以外使用内部类

定义类的主要作用就是定义变量、创建对象和派生子类。定义内部类的主要作用也如此。正如前面所看到的，在外部类内部使用嵌套类或内部类时，与平常使用普通类没有太大的区别，一样可以直接通过嵌套类或内部类的类名来定义变量，调用嵌套类或内部类的构造器来创建实例。

唯一要牢记的一点：嵌套类只能访问外部的其他嵌套类，不能访问外部的其他任何成员。

如果希望在外部类以外的地方使用内部类或嵌套类，一定要注意访问权限的限制，比如使用 private 修饰的嵌套类或内部类只能在外部类之内使用。

在外部类以外的地方定义内部类变量的语法格式如下：

```
var | val varName: OuterClass.InnerClass
```

从上面语法格式可以看出，在外部类以外的地方使用内部类时，内部类完整的类名应该是 OuterClass.InnerClass。如果外部类有包名，则还应该增加包名前缀。

由于内部类的对象必须寄生在外部类的对象中，因此在创建内部类对象之前，必须先创建其外部类对象。在外部类以外的地方创建内部类实例的语法格式如下：

```
OuterInstance.InnerConstructor()
```

从上面语法格式可以看出，在外部类以外的地方创建内部类实例时必须使用外部类实例来调用内部类的构造器。下面程序示范了如何在外部类以外的地方创建内部类对象，并把它赋值给内部类类型的变量。

程序清单：codes\08\8.5\CreateInnerInstance.kt

```
class Out{
    // 定义一个内部类，不使用访问控制符，默认是 public
    inner class In(msg: String){
        init{
            println(msg)
        }
    }
}
fun main(args: Array<String>) {
    var oi: Out.In = Out().In("测试信息")
    /*
    上面代码可改为如下三行代码
    使用 OutterClass.InnerClass 的形式定义内部类变量
    var oi: Out.In
    创建外部类实例，内部类实例将寄生在该实例中
    val ot = Out()
    通过外部类实例来调用内部类的构造器创建内部类实例
    oi = ot.In("测试信息")
    */
}
```

上面程序中粗体字代码创建了一个内部类对象。从上面代码可以看出，内部类的构造器必须使用外部类对象来调用。

8.5.4 在外部类以外使用嵌套类

因为嵌套类是属于外部类的类本身的，因此创建嵌套类对象时无须创建外部类对象，所以

嵌套类用起来非常方便。在外部类以外的地方创建嵌套类实例的语法格式如下：

```
OuterClass.NestedConstructor()
```

下面程序示范了如何在外部类以外的地方创建嵌套类实例。

程序清单：codes\08\8.5\CreateNestedInstance.kt

```
class NestedOut{
    // 定义一个嵌套类，不使用访问控制符，默认是 public
    open class Nested{
        init{
            println("嵌套类的构造器")
        }
    }
}
fun main(args: Array<String>) {
    val nested: NestedOut.Nested = NestedOut.Nested()
    /*
    上面代码可改为如下两行代码
    使用 OuterClass.NestedClass 的形式定义嵌套类变量
    val nested: NestedOut.Nested
    通过调用嵌套类的构造器创建嵌套类实例
    nested = NestedOut.Nested()
    */
}
```

从上面代码可以看出，不管是内部类还是嵌套类，其声明变量的语法完全一样。区别只是在创建对象时，嵌套类只需使用外部类即可调用构造器，而内部类必须使用外部类对象来调用构造器。

因为在调用嵌套类的构造器时无须使用外部类对象，所以创建嵌套类的子类也比较简单。下面代码就为嵌套类 NestedSubClass 定义了一个空的子类。

```
class NestedSubClass: NestedOut.Nested()
```

从上面代码可以看出，当定义一个嵌套类时，其外部类非常像一个包空间。

注意：

相比之下，使用嵌套类比使用内部类要简单很多，只要把外部类当作嵌套类的包空间即可。因此程序应该优先考虑使用嵌套类。



8.5.5 局部嵌套类

如果把一个嵌套类放在方法或函数中定义，则这个嵌套类就是一个局部嵌套类，局部嵌套类仅在该方法或函数中有效。由于局部嵌套类不能在方法或函数以外的地方使用，因此局部嵌套类也不能使用访问控制符修饰。

注意：

对于局部成员而言，不管是局部变量还是局部嵌套类，它们的上一级程序单元都是方法或函数。由于局部成员的作用域是所在方法或函数，其他程序单元永远也不可能访问另一个方法或函数中的局部成员，所以局部成员不能使用访问控制符修饰。



如果需要用局部嵌套类定义变量、创建实例或派生子类，那么都只能在局部嵌套类所在的方法（或函数）内进行。

程序清单：codes\08\8.5\LocalNestedClass.kt

```
class LocalNestedClass{
    fun info(){
        // 定义局部嵌套类
        open class NestedBase(var a: Int = 0){
        }
        // 定义局部嵌套类的子类
        class NestedSub(var b: Int = 0): NestedBase(){
        }
        // 创建局部嵌套类的对象
        val ns = NestedSub()
        ns.a = 5
        ns.b = 8
        println("NestedSub 对象的 a 和 b 属性是：${ns.a},${ns.b}")
    }
}
fun main(args: Array<String>) {
    LocalNestedClass().info()
}
```

编译上面程序，可以看到生成了三个 class 文件：LocalNestedClass.class、LocalNestedClass$1-NestedBase.class 和 LocalNestedClass$1NestedSub.class，这表明局部嵌套类的 class 文件总是遵循如下命名格式：OuterClass$NNestedClass.class。注意到局部嵌套类的 class 文件的文件名比嵌套类、内部类的 class 文件的文件名多了一个数字，这是因为同一个类中不可能有两个同名的嵌套类、内部类，而同一个类中则可能有两个以上同名的局部嵌套类（处于不同的方法中），所以 Kotlin 为局部嵌套类的 class 文件名增加了一个数字，用于区分。



提示：

局部嵌套类是一个非常“鸡肋”的语法，在实际开发中很少定义局部嵌套类，这是因为局部嵌套类的作用域太小了，只能在当前方法中使用。大部分时候，定义一个类之后，当然希望多次复用这个类，但局部嵌套类无法离开它所在的方法，因此在实际开发中很少使用局部嵌套类。

8.5.6 匿名内部类

Java 有一个非常实用的功能：匿名内部类，Kotlin 则彻底抛弃了这个功能。不过读者不用担心，Kotlin 提供了一个更加强大的语法：对象表达式。



提示：

对象表达式其实就是增强版的匿名内部类，下一节会详细介绍。

此外，如果对象是函数式接口（只包含一个抽象方法的接口）的实例，则可使用带接口类型前缀的 Lambda 表达式创建它。例如，如下代码创建了一个 Runnable 实例来启动线程。

程序清单：codes\08\8.5\RunnableTest.kt

```
fun main(args: Array<String>) {
    // 使用 Lambda 表达式创建 Runnable 实例
    var t = Runnable{
        for (i in 0..100) {
            println("${Thread.currentThread().getName()},i: ${i}")
        }
    }
    // 启动新线程
    Thread(t).start()
    // 主线程的循环
    for (i in 0..100) {
        println("${Thread.currentThread().getName()},i: ${i}")
    }
}
```

上面程序中粗体字代码使用 Lambda 表达式创建了一个 Runnable 实例，通过 Runnable 实例可启动多个线程。

8.6 对象表达式和对象声明

Kotlin 提供了比匿名内部类更加强大的语法：对象表达式。它们的主要区别在于：匿名内部类只能指定一个父类型（接口或父类），但对象表达式可指定 0~*N* 个父类型（接口或父类）。

8.6.1 对象表达式

对象表达式的语法格式如下：

```
object[: 0~N 个父类型] {
    // 对象表达式的类体部分
}
```

从上面语法格式可以看出，对象表达式可以指定 0~*N* 个父类型（类或接口），对象表达式的本质就是增强版的匿名内部类，因此编译器处理对象表达式时也会像匿名内部类一样生成对应的 class 文件。

关于对象表达式还有如下规则。

- 对象表达式不能是抽象类，因为系统在创建对象表达式时会立即创建对象。因此不允许将对象表达式定义成抽象类。
- 对象表达式不能定义构造器。但对象表达式可以定义初始化块，可以通过初始化块来完成构造器需要完成的事情。
- 对象表达式可以包含内部类（有 inner 修饰的内部类），不能包含嵌套类。

下面程序示范了几种对象表达式。

程序清单：codes\08\8.6\ObjectExprTest.kt

```
interface Outputable{
    fun output(msg: String)
}
abstract class Product(var price: Double){
    abstract val name: String
    abstract fun printInfo()
```

```
}
fun main(args: Array<String>) {
    // 指定一个父类型（接口）的对象表达式
    var ob1 = object: Outputable{
        // 重写父接口中的抽象方法
        override fun output(msg: String){
            for(i in 1..6){
                println("<h${i}>${msg}</h${i}>")
            }
        }
    }
    ob1.output("疯狂软件教育中心")
    println("------------------------")
    // 指定零个父类型的对象表达式
    var ob2 = object {
        // 初始化块
        init{
            println("初始化块")
        }
        // 属性
        var name = "Kotlin"
        // 方法
        fun test(){
            println("test 方法")
        }
        // 只能包含内部类，不能包含嵌套类
        inner class Foo
    }
    println(ob2.name)
    ob2.test()
    println("------------------------")
    // 指定两个父类型的对象表达式
    // 由于 Product 只有一个带参数的构造器，因此需要传入构造器参数
    var ob3 = object: Outputable, Product(28.8){
        override fun output(msg: String){
            println("输出信息：" + msg)
        }
        override val name: String
            get() = "激光打印机"
        override fun printInfo(){
            println("高速激光打印机，支持自动双面打印！")
        }
    }
    println(ob3.name)
    ob3.output("Kotlin 真不错！")
    ob3.printInfo()
}
```

上面程序中第一行粗体字代码的对象表达式只有一个父类型（接口），这个对象表达式与 Java 的匿名内部类实现太像了，除了 Kotlin 的对象表达式需要使用 object 关键字，该对象表达式定义的匿名类实现了 Outputable 接口，因此在类体部分实现了该接口中的抽象方法。

从上面介绍可以看出，Kotlin 的对象表达式可以完全取代 Java 的匿名内部类。

上面程序中第二行粗体字代码的对象表达式则没有指定父类型，这就是对象表达式的增强之处。我们在该对象表达式的类体部分定义了初始化块（程序创建对象时会自动执行）、属性、

方法、内部类，由此可见，对象表达式的本质其实就是增强版的匿名内部类，因此在类体部分除不能定义构造器、嵌套类之外，完全可以定义其他成员。

上面程序中第三行粗体字代码的对象表达式指定了两个父类型，一个是接口，一个是抽象类，这也是对象表达式的增强之处。此处需要说明的是，就像 Java 的匿名内部类继承抽象类时，必须调用抽象类的指定构造器一样，Kotlin 也是如此，所以我们使用了 Product(28.8)，表明调用 Product 抽象类的带一个 Double 参数的构造器。该对象表达式的类体部分没有任何特别之处，它只是重写了接口、抽象父类的抽象成员。



提示：

对于一个熟悉 Java 的读者而言，掌握对象表达式非常简单，只要牢记一点：Kotlin 的对象表达式就是匿名内部类的增强版，它支持指定 0～*N* 个父类型。

编译、运行上面程序，可看到如下输出：

```
<h1>疯狂软件教育中心</h1>
<h2>疯狂软件教育中心</h2>
<h3>疯狂软件教育中心</h3>
<h4>疯狂软件教育中心</h4>
<h5>疯狂软件教育中心</h5>
<h6>疯狂软件教育中心</h6>
-------------------------
初始化块
Kotlin
test 方法
-------------------------
激光打印机
输出信息：Kotlin 真不错！
高速激光打印机，支持自动双面打印！
```

仔细查看上面的输出，细心的读者应该会发现一个特征：ob2 对象没有继承任何父类型，程序为该对象（相当于匿名内部类）定义了 name 属性和 test()方法，接下来程序居然可以直接调用该对象的 name 属性和 test()方法——这对 Java 的匿名内部类来说是不可想象的：Java 为匿名内部类增加的方法几乎无法直接访问，因为编译器只会把匿名内部类当成它所继承的父类或所实现的接口处理。

但 Kotlin 的对象表达式不同，Kotlin 的对象表达式可分为两种情形。

- 对象表达式在方法（或函数）的局部范围内，或使用 private 修饰的对象表达式，Kotlin 编译器可识别该对象表达式的真实类型，就像上面的代码所示：程序为 ob2 增加了方法和属性，在 main()函数的局部范围内，Kotlin 编译器完全可以识别 ob2 的真实类型，因此 ob2 可调用对象表达式增加的属性和方法。
- 非 private 修饰的对象表达式与 Java 的匿名内部类相似，编译器只会把对象表达式当成它所继承的父类或所实现的接口处理。如果它没有父类型，系统当它是 Any 类型。

如下程序示范了 Kotlin 编译器处理对象表达式类型的两种情形。

程序清单：codes\08\8.6\ObjectExprType.kt

```
class ObjectExprType{
    private val ob1 = object {
        val name: String = "fkit"
    }
    internal val ob2 = object {
```

```
        val name: String = "fkit"
    }
    private fun privateBar() = object {
        val name: String = "fkjava.org"
    }
    fun publicBar() = object {
        val name: String = "fkjava.org"
    }
    fun test(){
        // ob1 是 private 对象表达式，编译器可识别它的真实类型
        // 下面代码正确
        println(ob1.name)
        // ob2 是非 private 对象表达式，编译器当它是 Any 类型
        // 下面代码错误
//        println(ob2.name)
        // privateBar 是 private 函数，编译器可识别它返回的对象表达式的真实类型
        // 下面代码正确
        println(privateBar().name)
        // publicBar 是非 private 函数，编译器将它返回的对象表达式当成 Any 类型
        // 下面代码错误
//        println(publicBar().name)
    }
}
fun main(args: Array<String>) {
    ObjectExprType().test()
}
```

从上面代码可以看出，Kotlin 编译器可识别 private 对象表达式的真实类型。

此外，Kotlin 的对象表达式可访问或修改其作用域内的局部变量（Java 只能访问其所在范围内的 effectively final 局部变量）。

例如如下程序。

程序清单：codes\08\8.6\ObjectExprLocal.kt

```
fun main(args: Array<String>) {
    var a = 20
    var obj = object {
        fun change(){
            println("change()方法修改变量 a 的值")
            a++
        }
    }
    obj.change()
    println(a) // 输出 21
}
```

总结起来，Kotlin 的对象表达式比 Java 的匿名内部类增强了三个方面。

- 对象表达式可指定多个父类型。
- Kotlin 编译器能更准确地识别局部范围内 private 对象表达式的类型。
- 对象表达式可访问或修改其所在范围内的局部变量。

8.6.2 对象声明和单例模式

对象声明的语法格式如下：

```
object ObjectName[: 0~N 个父类型] {
    // 对象表达式的类体部分
}
```

从上面语法格式可以看出，对象声明与对象表达式的语法非常相似，似乎它们之间唯一的

区别是：对象表达式在object关键字后没有名字；而对象声明需要在object关键字后指定名字。

实际上，对象声明和对象表达式还存在如下区别。

- 对象表达式是一个表达式，因此它可以被赋值给变量；而对象声明不是表达式，因此它不能用于赋值。
- 对象声明可包含嵌套类，不能包含内部类；而对象表达式可包含内部类，不能包含嵌套类。
- 对象声明不能定义在函数和方法内；但对象表达式可嵌套在其他对象声明或非内部类中。

下面我们用对象声明来改写上一节中对象表达式的第一个程序。

程序清单：codes\08\8.6\ObjectDeclTest.kt

```
interface Outputable{
    fun output(msg: String)
}
abstract class Product(var price: Double){
    abstract val name: String
    abstract fun printInfo()
}
// 指定一个父类型（接口）的对象表达式
object MyObject1: Outputable{
    // 重写父接口中的抽象方法
    override fun output(msg: String){
        for(i in 1..6){
            println("<h${i}>${msg}</h${i}>")
        }
    }
}
// 指定零个父类型的对象表达式
object MyObject2 {
    // 初始化块
    init{
        println("初始化块")
    }
    // 属性
    var name = "Kotlin"
    // 方法
    fun test(){
        println("test 方法")
    }
    // 只能包含嵌套类，不能包含内部类
    class Foo
}
// 指定两个父类型的对象表达式
// 由于 Product 只有一个带参数的构造器，因此需要传入构造器参数
object MyObject3: Outputable, Product(28.8){
    override fun output(msg: String){
        println("输出信息：" + msg)
    }
    override val name: String
```

```
        get() = "激光打印机"
    override fun printInfo(){
        println("高速激光打印机，支持自动双面打印！")
    }
}
fun main(args: Array<String>) {
    MyObject1.output("疯狂软件教育中心")
    println("-------------------------")
    println(MyObject2.name)
    MyObject2.test()
    println("-------------------------")
    println(MyObject3.name)
    MyObject3.output("Kotlin 真不错！")
    MyObject3.printInfo()
}
```

将上面程序与上一节中对象表达式的第一个程序进行对比，即可发现对象声明不能用于赋值，对象声明本身已有名称，因此可以使用对象声明的名称访问该对象。此外，程序还将对象声明移到 main()函数之外，这是因为对象声明不允许放在函数或方法内。

对象声明专门用于实现单例模式，对象声明所定义的对象也就是该类的唯一实例，程序可通过对象声明的名称直接访问该类的唯一实例。

8.6.3 伴生对象和静态成员

在类中定义的对象声明，可使用 companion 修饰，这样该对象就变成了伴生对象。

每个类最多只能定义一个伴生对象，伴生对象相当于外部类的对象，程序可通过外部类直接调用伴生对象的成员。例如如下代码。

程序清单：codes\08\8.6\CompanionTest.kt

```
interface Outputable{
    fun output(msg: String)
}
class MyClass{
    // 使用 companion 修饰的伴生对象
    companion object MyObject1: Outputable{
        val name = "name 属性值"
        // 重写父接口中的抽象方法
        override fun output(msg: String){
            for(i in 1..6){
                println("<h${i}>${msg}</h${i}>")
            }
        }
    }
}
fun main(args: Array<String>) {
    // 使用伴生对象所在的类调用伴生对象的方法
    MyClass.output("fkit.org")
    println(MyClass.name)
}
```

上面程序中第一行粗体字代码使用 companion 修饰了对象声明，因此该对象变成了伴生对象。接下来在主程序中即可使用伴生对象所在的类调用伴生对象的成员。



提示：

从上面的 MyClass.output("fkit.org")代码不难发现：这不就是 Java 使用类访问静态成员的语法吗？实际上确实如此，由于 Kotlin 取消了 static 关键字，因此 Kotlin 引入伴生对象来弥补没有静态成员的不足。可见，伴生对象的主要作用就是为其所在的外部类模拟静态成员。

从上面程序中第二行粗体字代码可以看出，伴生对象的名称并不重要，因此伴生对象可以省略名称。省略名称之后，如果程序真的要访问伴生对象，则可通过 Companion 名称进行访问。例如如下代码。

程序清单：codes\08\8.6\CompanionNoName.kt

```
interface Outputable{
    fun output(msg: String)
}
class MyClass{
    // 省略名字的伴生对象
    companion object: Outputable{
        // 重写父接口中的抽象方法
        override fun output(msg: String){
            for(i in 1..6){
                println("<h${i}>${msg}</h${i}>")
            }
        }
    }
}
fun main(args: Array<String>) {
    // 使用伴生对象所在的类调用伴生对象的方法
    MyClass.output("fkit.org")
    // 使用 Companion 名称访问伴生对象
    println(MyClass.Companion)
}
```

虽然伴生对象的主要作用就是为它所在的类模拟静态成员，但只是模拟，伴生对象的成员依然是伴生对象本身的实例成员，并不属于伴生对象所在的外部类。



提示：

在 JVM 平台上，可通过@JvmStatic 注解让系统根据伴生对象的成员为其所在的外部类生成真正的静态成员。具体信息可参考本书第 12 章。

8.6.4 伴生对象的扩展

伴生对象也可以被扩展。如果一个类具有伴生对象，则 Kotlin 允许为伴生对象扩展方法和属性。为伴生对象扩展的方法和属性，就相当于为伴生对象所在的外部类扩展了静态成员，可通过外部类的类名访问这些扩展成员。

如下程序示范了为伴生对象扩展成员。

程序清单：codes\08\8.6\CompanionExtend.kt

```
interface Outputable{
    fun output(msg: String)
```

```
}
class MyClass{
    // 使用 companion 修饰的伴生对象
    companion object: Outputable{
        val name = "name 属性值"
        // 重写父接口中的抽象方法
        override fun output(msg: String){
            for(i in 1..6){
                println("<h${i}>${msg}</h${i}>")
            }
        }
    }
}
// 为伴生对象扩展方法
fun MyClass.Companion.test(){
    println("为伴生对象扩展的方法")
}
val MyClass.Companion.foo
    get() = "为伴生对象扩展的属性"
fun main(args: Array<String>) {
    // 使用伴生对象所在的类调用伴生对象的成员
    MyClass.output("fkit.org")
    println(MyClass.name)
    // 通过伴生对象所在的类调用为伴生对象扩展的成员
    MyClass.test()
    println(MyClass.foo)
}
```

上面程序中两行粗体字代码示范了为伴生对象扩展方法和属性，正如前面所提到的，Kotlin 可通过 Companion 名称访问伴生对象，因此这两行粗体字代码显式指定为 MyClass.Companion 扩展方法和属性，这就是为伴生对象扩展方法和属性。

正如从上面 main()函数中两行粗体字代码所看到的，为伴生对象扩展成员之后，伴生对象的外部类也可像访问自己的类成员一样访问它们。

8.7 枚举类

Kotlin 当然也支持枚举类，Kotlin 的枚举类与 Java 差别不大。

8.7.1 枚举类入门

Kotlin 使用 enum class 关键字组合定义枚举类。枚举类是一种特殊的类，它一样可以有自己的属性、方法，可以实现一个或多个接口，也可以定义自己的构造器。

但枚举类与普通类有如下简单区别。

- 枚举类可以实现一个或多个接口，使用 enum 定义的枚举类默认继承 kotlin.Enum 类，而不是默认继承 Any 类，因此枚举类不能显式继承其他父类。其中 Enum 类实现了 kotlin.Comparable 接口。
- 使用 enum 定义的非抽象的枚举类不能使用 open 修饰，因此枚举类不能派生子类。
- 枚举类的构造器只能使用 private 访问控制符，如果省略了构造器的访问控制符，则默认使用 private 修饰；如果强制指定访问控制符，则只能指定 private 修饰符。

- 枚举类的所有实例必须在枚举类的第一行显式列出，否则这个枚举类永远都不能产生实例。列出枚举实例后最好用分号结尾。

枚举类默认提供了如下两个方法。

- EnumClass.valueOf(value: String): EnumClass：类似于 Java 枚举类的 valueOf()方法，用于根据枚举的字符串名获取实际的枚举值。如果传入的名称参数与类中定义的任何枚举常量均不匹配，valueOf()方法将抛出 IllegalArgumentException 异常。
- EnumClass.values(): Array<EnumClass>：类似于 Java 枚举类的 values()方法，用于获取该枚举的所有枚举值组成的数组。

下面程序定义了一个 Season 枚举类。

程序清单：codes\08\8.7\Season.kt

```
enum class Season{
    // 在第一行列出 4 个枚举实例
    SPRING, SUMMER, FALL, WINTER
}
```

编译上面的 Kotlin 程序，将生成一个 Season.class 文件，这表明枚举类是一种特殊的类。由此可见，enum class 关键字组合和 class、interface 关键字的作用大致相似。

在定义枚举类时，需要显式列出所有的枚举值，如上面的 SPRING, SUMMER, FALL, WINTER 所示，所有的枚举值之间以英文逗号（,）隔开。这些枚举值代表了该枚举类的所有可能的实例。

如果需要使用该枚举类的某个实例，则可使用 EnumClass.variable 的形式，如 Season.SPRING。

程序清单：codes\08\8.7\EnumTest.kt

```
fun main(args: Array<String>) {
    // 枚举类默认有一个 values()方法，返回该枚举类的所有实例
    for (s in Season.values()){
        println(s)
    }
    val seasonName = "SUMMER"
    val s: Season = Season.valueOf(seasonName)
    println(s)
    // 直接访问枚举值
    println(Season.WINTER)
}
```

上面程序测试了 Season 枚举类的用法，该类通过 values()方法返回了 Season 枚举类的所有实例，并通过循环迭代输出了 Season 枚举类的所有实例；还使用了 Season 的 valueOf()方法根据字符串参数来获取枚举值。

前面已经介绍过，所有的枚举类都继承了 kotlin.Enum 类，所以枚举类可以直接使用该类中所包含的属性和方法。kotlin.Enum 类中提供了如下属性和方法。

- name 属性：返回此枚举实例的名称，这个名称就是定义枚举类时列出的所有枚举值之一。与此属性相比，大多数程序员应该优先考虑使用 toString()方法，因为 toString()方法可返回对用户更加友好的名称。
- ordinal 属性：返回枚举值在枚举类中的索引值（就是枚举值在枚举声明中的位置，第一个枚举值的索引值为 0）。

- int compareTo(E o)：该方法用于与指定的枚举对象比较顺序，同一个枚举实例只能与相同类型的枚举实例进行比较。如果该枚举对象位于指定的枚举对象之后，则返回正整数；如果该枚举对象位于指定的枚举对象之前，则返回负整数；否则返回 0。
- String toString()：返回枚举常量的名称，与 name 属性相似，但 toString()方法更常用。

正如前面所看到的，当程序使用 println(s)语句来打印枚举值时，实际上输出的是该枚举值 toString()方法的返回值，也就是输出该枚举值的名字。

自 Kotlin 1.1 开始，枚举可使用 enumValues<T>()和 enumValueOf<T>()函数以泛型的方式访问枚举类中的常量，因此可为所有枚举定义如下方法来遍历所有的枚举实例。

```
inline fun <reified T : Enum<T>> printAllValues() {
    println(enumValues<T>().joinToString { it.name })
}
```

接下来程序按如下方式调用该函数输出所有枚举值。

```
printAllValues<Season>() // 输出 SPRING, SUMMER, FALL, WINTER
```



提示：

上面函数使用了泛型和 reified 关键字，关于该关键字的用法请参考本书第 10 章中关于泛型的介绍。

8.7.2 枚举类的属性、方法和构造器

枚举类也是一种类，只是一种比较特殊的类，因此它一样可以定义属性、方法和构造器。

由于枚举类应该设计成不可变类（具体原因可参考《疯狂 Java 讲义》），因此它的属性值不允许改变，这样会更安全。Kotlin 禁止开发者对属性赋值，并推荐使用 val 为枚举声明只读属性。

由于枚举的属性都是只读属性，枚举必须在构造器中为这些属性指定初始值（或在初始化块中指定初始值，一般不会在定义时指定初始值，因为这样会导致所有枚举值的该属性值总是相同的），因此应该为枚举类显式定义带参数的构造器。

一旦为枚举类显式定义了带参数的构造器，在列出枚举值时就必须对应地传入参数。例如如下程序。

程序清单：codes\08\8.7\Gender.kt

```
// 使用主构造器声明 cnName 只读属性
enum class Gender(val cnName: String) {
    MALE("男"), FEMALE("女");
    // 定义方法
    fun info(){
        when(this) {
            MALE -> println("天行健，君子以自强不息")
            FEMALE -> println("地势坤，君子以厚德载物")
        }
    }
}
```

从上面程序中可以看出，当为 Gender 枚举类定义了一个 Gender(String)构造器之后，列出枚举值就应该采用粗体字代码来完成。也就是说，在枚举类中列出枚举值时，实际上就是调用

构造器创建枚举类对象，只是这里无须显式调用构造器。前面列出枚举值时无须传入参数，甚至无须使用括号，仅仅是因为前面的枚举类包含无参数的构造器。

不难看出，上面程序中的粗体字代码实际上等同于如下代码：

```
MALE = new Gender("男"), FEMALE = new Gender("女");
```

此外，由于上面的枚举类需要在列出枚举值之后定义额外的枚举成员（如枚举方法），因此上面程序需要用分号表示枚举值列表结束。

下面程序示范了 Gender 枚举类的用法。

程序清单：codes\08\8.7\GenderTest.kt

```
fun main(args: Array<String>) {
    // 通过 Gender 的 valueOf()方法根据枚举名获取枚举值
    val g = Gender.valueOf("FEMALE")
    // 访问枚举值的 cnName 属性
    println("${g}代表:${g.cnName}")
    // 调用 info 方法
    g.info()
}
```

上面两行粗体字代码示范了访问枚举的属性和方法。

8.7.3 实现接口的枚举类

枚举类也可以实现一个或多个接口。与普通类实现一个或多个接口完全一样，枚举类实现一个或多个接口时，也需要实现该接口所包含的方法。下面程序定义了一个 GenderDesc 接口。

程序清单：codes\08\8.7\interface\GenderDesc.kt

```
interface GenderDesc{
    fun info()
}
```

在上面的 GenderDesc 接口中定义了一个 info()方法，下面的 Gender 枚举类实现了该接口，并实现了该接口中包含的 info()方法。

程序清单：codes\08\8.7\interface\Gender.kt

```
// 使用主构造器声明 cnName 只读属性
enum class Gender(val cnName: String): GenderDesc {
    MALE("男"), FEMALE("女");
    // 重写接口中定义的抽象方法
    override fun info(){
        when(this) {
            MALE -> println("天行健，君子以自强不息")
            FEMALE -> println("地势坤，君子以厚德载物")
        }
    }
}
```

读者可能会发现，枚举类实现接口不过如此，与普通类实现接口完全一样：同样要将被实现的接口放在英文冒号后面，并实现接口中包含的抽象方法。

8.7.4 包含抽象方法的抽象枚举类

假设有一个 Operation 枚举类，它的 4 个枚举值 PLUS, MINUS, TIMES, DIVIDE 分别代表

加、减、乘、除 4 种运算，该枚举类需要定义一个 eval()方法来完成计算。

从上面描述可以看出，Operation 需要让 PLUS、MINUS、TIMES、DIVIDE 这 4 个值对 eval()方法各有不同的实现。此时可考虑为 Operation 枚举类定义一个 eval()抽象方法，然后让 4 个枚举值分别为 eval()提供不同的实现。例如如下代码。

程序清单：codes\08\8.7\abstract\Operation.kt

```
enum class Operation{
    PLUS {
        override fun eval(x: Double, y: Double) = x + y
    },
    MINUS {
        override fun eval(x: Double, y: Double) = x - y
    },
    TIMES {
        override fun eval(x: Double, y: Double) = x * y
    },
    DIVIDE {
        override fun eval(x: Double, y: Double) = x / y
    };
    // 为枚举类定义一个抽象方法
    // 这个抽象方法由不同的枚举值提供不同的实现
    abstract fun eval(x: Double, y: Double): Double
}
fun main(args: Array<String>) {
    println(Operation.PLUS.eval(3.0, 4.0))
    println(Operation.MINUS.eval(5.0, 4.0))
    println(Operation.TIMES.eval(5.0, 4.0))
    println(Operation.DIVIDE.eval(5.0, 4.0))
}
```

编译上面程序，会生成 5 个 class 文件，其中 Operation 对应一个 class 文件，它的 4 个匿名内部子类各对应一个 class 文件。

枚举类中包含抽象方法时依然不能使用 abstract 修饰枚举类（因为系统自动会为它添加 abstract 关键字），但因为枚举类需要显式创建枚举值，而不是作为父类，所以在定义每个枚举值时必须为抽象成员提供实现，否则将出现编译错误。

上面程序中的粗体字代码看起来有些奇怪：当创建 PLUS、MINUS、TIMES 和 DIVIDE 枚举值时，后面又紧跟了一对花括号，这对花括号中包含了一个 eval()方法定义。如果读者还记得对象表达式（或匿名内部类）语法的话，则可能就熟悉这样的语法了，花括号部分实际上就是一个类体部分，在这种情况下，当创建 PLUS、MINUS、TIMES 和 DIVIDE 枚举值时，并不是直接创建 Operation 枚举类的实例的，而是相当于创建 Operation 的匿名子类的实例。因为粗体字代码的括号部分实际上是对象表达式（匿名内部类）的类体部分，所以这个部分的代码语法与前面介绍的对象表达式（匿名内部类）的语法大致相似。

8.8 类委托和属性委托

委托是 Kotlin 的另一个特色功能，也是 Java 原本所不具备的功能。Kotlin 的委托可分为类委托和属性委托。

8.8.1 类委托

类委托是代理模式的应用，而代理模式可作为继承的一个不错的替代。类委托的本质就是将本类需要实现的部分方法委托给其他对象——相当于借用其他对象的方法作为自己的实现。

例如，下面定义一个类，该类实现了一个接口，但该类并不实现该接口中的抽象方法，而是借用其他对象中的方法来实现，被借用的对象被称为被委托对象。

程序清单：codes\08\8.8\ClassDelegation.kt

```
interface Outputable {
    fun output(msg: String)
    var type: String
}
// 定义一个 DefaultOutput 类实现 Outputable 接口
class DefaultOutput: Outputable{
    override fun output(msg: String){
        for (i in 1..6) {
            println("<h${i}>${msg}</h${i}>")
        }
    }
    override var type: String = "输出设备"
}
// 定义 Printer 类，指定构造参数 b 作为委托对象
class Printer(b: DefaultOutput): Outputable by b
// 定义 Projector 类，指定新建的对象作为委托对象
class Projector(): Outputable by DefaultOutput(){
    // 该类重写 output()方法
    override fun output(msg: String){
        javax.swing.JOptionPane.showMessageDialog(null, msg);
    }
}
fun main(args: Array<String>) {
    val output = DefaultOutput()
    // Printer 对象的委托对象是 output
    var printer = Printer(output)
     // 其实就是调用委托对象的 output()方法
    printer.output("fkit.org")
     // 其实就是调用委托对象的 type 属性方法
    println(printer.type)
    // Projector 对象的委托对象也是 output
    var projector = Projector()
    // Projector 本身重写了 output()方法，所以此处是调用本类重写的方法
    projector.output("疯狂软件教育中心")
    // 其实就是调用委托对象的 type 属性方法
    println(projector.type)
}
```

上面程序定义了 Printer 和 Projector 两个类，这两个类都实现了 Ouputable 接口，因此需要实现该接口中的抽象方法和抽象属性。程序通过 by 关键字为这两个类指定了委托对象，这意味着这两个类可直接“借用”被委托对象所实现的方法和属性。

上面程序中 Printer 类虽然没有实现 output()方法和 type 属性，但它的实例一样可以在 main()函数中调用 output()方法和 type 属性。Projector 类虽然指定了 DefaultOutput 对象作为委托，但由于该类本身也实现了 output()方法，因此当 Projector 对象调用 output()方法时，它不再需要调用委托对象的 output()方法，而是直接使用自己实现的方法。

由此可见，当一个类重写了委托对象所包含的方法之后，Kotlin 将优先使用该类自己实现的方法。

上面程序示范了为类指定委托对象的两种方式，第一种方式是通过构造器参数指定委托对象；第二种方式是直接在类定义的 by 后新建对象。通常会采用第一种方式，因为这样可以让多个对象共享同一个委托对象。

8.8.2 属性委托

Kotlin 也支持属性委托，属性委托可以将多个类的类似属性统一交给委托对象集中实现，这样就可避免每个类都需要单独实现这些属性。

对于指定了委托对象的属性而言，由于它的实现逻辑已经交给委托对象处理，因此开发者不能为委托属性提供 getter 和 setter 方法，Kotlin 也不会为委托属性提供 getter、setter 方法的默认实现。

一旦将某个对象指定为属性的委托对象，该对象就会全面接管该属性的读取（getter）和写入（setter）操作，因此属性的委托对象无须实现任何接口，但一定要提供一个 getValue()方法和 setValue()方法（val 属性无须提供 setter 方法）。

对于只读属性（用 val 声明的属性）而言，由于程序无须设置属性值，只要能读取属性值即可，因此只读属性的委托对象只需提供使用 operator 修饰的 getValue()方法。该方法的参数和返回值要求如下。

- thisRef：该参数代表属性所属的对象，因此该参数的类型必须是属性所属对象的类型（对于扩展属性，指被扩展的类型）或其超类型。
- property：该参数代表目标属性，该参数的类型必须是 KProperty<*>或其超类型。
- 返回值：该方法必须返回与目标属性相同的类型（或其子类型）。

对于读写属性（用 var 声明的属性）而言，属性的委托对象必须额外提供一个使用 operator 修饰的 setValue()方法，该方法不需要返回值，但必须指定如下参数。

- thisRef：与 getValue()的 thisRef 参数的作用相同。
- property：与 getValue()的 property 参数的作用相同。
- newValue：该参数代表目标属性新设置的属性值，该参数的类型必须具有和目标属性相同的类型或其超类型。

属性的委托对象的 getValue()、setValue()方法既可以是成员方法，也可以是扩展方法。总之，只要是符合规则的两个方法均可。



提示：

Kotlin 标准库在 kotlin.properties 下提供了 ReadOnlyProperty 和 ReadWriteProperty 两个接口，其中 ReadOnlyProperty 接口定义了一个符合只读属性委托标准的 getValue()抽象方法，因此该接口的实现类可作为只读属性的委托对象；而 ReadWriteProperty 接口定义了符合读写属性委托标准的 getValue()和 setValue()抽象方法，因此该接口的实现类可作为读写属性的委托对象。

下面程序示范了属性委托的用法。

程序清单：codes\08\8.8\PropertyDelegation.kt

```
import kotlin.reflect.*;
```

```
class PropertyDelegation{
    // 该属性的委托对象是 MyDelegation
    var name: String by MyDelegation()
}
class MyDelegation{
    private var _backValue = "默认值"
    operator fun getValue(thisRef: PropertyDelegation,
        property: KProperty<*>): String {
        println("${thisRef}的${property.name}属性执行 getter 方法")
        return _backValue
    }
    operator fun setValue(thisRef: PropertyDelegation,
        property: KProperty<*>, newValue: String) {
        println("${thisRef}的${property.name}属性执行 setter 方法" +
            ", 传入参数值为: ${newValue}")
        _backValue = newValue
    }
}
fun main(args: Array<String>) {
    val pd = PropertyDelegation()
    // 读取属性，实际上是调用属性的委托对象的 getter 方法
    println(pd.name)
    // 写入属性，实际上是调用属性的委托对象的 setter 方法
    pd.name = "fkit.org"
    println(pd.name)
}
```

上面程序中粗体字代码将 PropertyDelegation 的 name 属性委托给 MyDelegation 对象。接下来程序定义了 MyDelegation 类，由于该类的对象要作为读写属性的委托对象，因此该类必须实现符合要求的 getter、setter 方法，如 MyDelegation 中粗体字代码所示。

当程序读取 PropertyDelegation 对象的 name 属性时，实际上就是执行 MyDelegation 对象所提供的 getValue()方法；当程序对 PropertyDelegation 对象的 name 属性赋值时，实际上就是执行 MyDelegation 对象所提供的 setValue()方法。

编译、运行上面程序，可看到如下输出：

```
PropertyDelegation@1de0aca6 的 name 属性执行 getter 方法
默认值
PropertyDelegation@1de0aca6 的 name 属性执行 setter 方法，传入参数值为：fkit.org
PropertyDelegation@1de0aca6 的 name 属性执行 getter 方法
```



提示：

实际上，上面的 MyDelegation 类已经实现了 ReadWriteProperty 接口的规范，因此程序完全可让该类实现 ReadWriteProperty<PropertyDelegation, String>接口，再为 getValue()和 setValue()方法添加 override 修饰符。

前面已经讲过，Kotlin 不会为委托属性生成幕后字段、getter 和 setter 方法——这是不言而喻的，因此委托属性的所有处理细节都由委托对象负责。

Kotlin 会委托属性生成辅助属性，该辅助属性引用了属性的委托对象。当程序调用委托属性的 getter、setter 方法时，实际上就是执行委托对象的 getValue()、setValue()方法。

例如，我们定义了如下委托属性：

```
class C {
    var prop: Type by MyDelegate()
}
```

上面代码定义了名为 prop 的委托属性，该属性的委托对象是 MyDelegate。上面代码被编译器处理之后将生成如下代码：

```
// 这段代码是由编译器生成的
class C {
    // 生成隐藏属性，隐藏属性引用代理对象
    private val prop$delegate = MyDelegate()
    var prop: Type
        // getter 方法，实际上就是调用代理对象的 getValue() 方法
        get() = prop$delegate.getValue(this, this::prop)
        // setter 方法，实际上就是调用代理对象的 setValue() 方法
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

在上面编译器生成的 getter、setter 方法实现中，其实就是调用代理对象的 getValue()、setValue()方法，传入的第一个参数 this 代表正在调用 getter 或 setter 方法，第二个参数 this::prop 代表正在操作的目标属性。

8.8.3 延迟属性

Kotlin 提供了一个 lazy()函数，该函数接受一个 Lambda 表达式作为参数，并返回一个 Lazy<T>对象。

Lazy<T>对象包含了一个符合只读属性委托要求的 getValue()方法，因此该 Lazy<T>对象可作为只读属性的委托对象。

注意：

Lazy<T>对象只能作为只读属性的委托对象。



例如如下代码。

程序清单：codes\08\8.8\LazyPropertyTest.kt

```
val lazyProp: String by lazy {
    println("第一次访问时执行代码块")
    "疯狂软件"
}
fun main(args: Array<String>) {
    // 两次访问 lazyProp 属性
    println(lazyProp)
    println(lazyProp)
}
```

上面程序中粗体字代码将 lazyProp 属性的委托对象指定为 lazy()函数的返回值：Lazy<T>对象。这样当程序访问 lazyProp 属性时，实际上就是调用 Lazy<T>对象的 getValue()方法。

Lazy<T>的 getValue()方法的处理逻辑是：第一次调用该方法时，程序会计算 Lambda 表达式，并得到其返回值，以后程序再次调用该方法时，不再计算 Lambda 表达式，而是直接使用第一次计算得到的返回值。

运行上面程序，可看到如下输出：

```
第一次访问时执行代码块
疯狂软件
疯狂软件
```

lazy()函数有如下两个版本。

- fun <T> lazy(initializer: () -> T): Lazy<T>
- fun <T> lazy(mode: LazyThreadSafetyMode, initializer: () -> T): Lazy<T>

从上面代码可以看出，前面代码使用的是第一个版本，该版本的 lazy()函数执行 Lambda 表达式时会提供自动添加保证线程安全的同步锁，因此开销略大。



提示：

上面的第一个 lazy()函数相当于第二个 lazy()函数的第一个参数指定为 LazyThreadSafetyMode.SYNCHRONIZED。

如果程序不需要 lazy()函数保证线程安全，希望提供更好的性能，则可将 lazy()函数的第一个参数指定为如下两种模式。

- LazyThreadSafetyMode.PUBLICATION：在这种模式下，lazy()函数不会使用排他同步锁，多个线程可同时执行。
- LazyThreadSafetyMode.NONE：在这种模式下，lazy()函数不会有任何线程安全相关的操作和开销。

例如，将上面的 lazyProp 改为如下形式来取消线程安全保证，从而获得最佳性能（程序清单同上）。

```
val lazyProp: String by lazy(LazyThreadSafetyMode.NONE) {
    println("第一次访问时执行代码块")
    "疯狂软件"
}
```

8.8.4 属性监听

Kotlin 的 kotlin.properties 包下提供了一个 Delegates 对象声明（单例对象），该对象中包含如下两个常用方法。

- fun <T> observable(initialValue: T, onChange: (property: KProperty<*>, oldValue: T, newValue: T) -> Unit): ReadWriteProperty<Any?, T>：该方法返回一个 ReadWriteProperty 对象，因此该方法的返回值可作为只读属性的委托对象。该方法的第一个参数用于为接受委托的属性指定初始值，第二个参数可接受 Lambda 表达式，当接受委托的属性被设置值时，该 Lambda 表达式就会被触发。



提示：

根据该方法的描述不难发现，observable()方法返回的 ReadWriteProperty 对象其实就是属性监听器，只要该属性被重新设置值，observable()方法的 Lambda 参数就会被执行。

- fun <T> vetoable(initialValue: T, onChange: (property: KProperty<*>, oldValue: T, newValue: T) -> Boolean): ReadWriteProperty<Any?, T>：该方法与前一个方法基本相同，区别只是负责执行监听的 Lambda 表达式需要返回值，当该返回值为 true 时，接受委托的属性的新值设置成功；否则设置失败。

例如如下程序。

程序清单：codes\08\8.8\ObservablePropertyTest.kt

```
import kotlin.properties.*;
var observableProp: String by Delegates.observable("默认值") {
        // Lambda 表达式的第一个参数代表被监听的属性
        // 第二个参数代表修改之前的值
        // 第三个参数代表被设置的新值
        prop, old, new ->
        println("${prop} 的 ${old} 被改为 ${new} ")
    }
fun main(args: Array<String>) {
    // 访问 observableProp 属性，不会触发监听器的 Lambda 表达式
    println(observableProp)
    // 设置属性值，触发监听器的 Lambda 表达式
    observableProp = "疯狂软件"
    println(observableProp)
}
```

运行上面程序，可看到如下输出：

```
默认值
var observableProp: kotlin.String 的默认值被改为疯狂软件
疯狂软件
```

从上面的运行结果可以看出，无论我们为属性设置的新值是什么，总可以设置成功。如果需要对新值进行判断，然后再确定是否设置成功，则可使用 Delegates 对象的 vetoable()方法。例如如下代码。

程序清单：codes\08\8.8\Vetoable PropertyTest.kt

```
import kotlin.properties.*;
var vetoableProp: Int by Delegates.vetoable(20) {
        // Lambda 表达式的第一个参数代表被监听的属性
        // 第二个参数代表修改之前的值
        // 第三个参数代表被设置的新值
        prop, old, new ->
            println("${prop} 的 ${old} 被改为 ${new} ")
            new > old
    }
fun main(args: Array<String>) {
    // 访问 vetoableProp 属性，不会触发监听器的 Lambda 表达式
    println(vetoableProp)
    // 新值小于旧值，监听的 Lambda 表达式返回 false，新值设置失败
    vetoableProp = 15
    println(vetoableProp) // 输出 20
    // 新值大于旧值，监听的 Lambda 表达式返回 true，新值设置成功
    vetoableProp = 25
    println(vetoableProp) // 输出 25
}
```

上面 Lambda 表达式内最后的表达式就是它的返回值，如果 new > old，则返回 true；否则返回 false。由此可见，此处我们要求新设置的属性值必须比原属性值大才能设置成功。

编译、运行程序，可看到如下输出：

```
20
var vetoableProp: kotlin.Int 的 20 被改为 15
```

```
20
var vetoableProp: kotlin.Int 的 20 被改为 25
25
```



提示：

现代编程语言几乎都提供了属性监听的功能，由于 Java 并没有真正意义上的属性（Java 的属性等于 field、getter、setter 方法的组合），因此 Java 对属性的监听需要通过 setter 方法来实现。而 Kotlin 的属性监听则通过属性委托机制来实现。

8.8.5 使用 Map 存储属性值

Kotlin 的 Map 提供了如下方法。

- operator fun <V, V1 : V> Map<in String, V>.getValue(thisRef: Any?, property: KProperty<*>): V1：该方法符合只读属性的委托对象的要求，这意味着 Map 对象可作为只读对象的委托。

MutableMap 则提供如下两个方法。

- operator fun <V, V1 : V> Map<in String, V>.getValue(thisRef: Any?, property: KProperty<*>): V1
- operator fun <V> MutableMap<in String, in V>.setValue(thisRef: Any?, property: KProperty<*>, value: V)

上面两个方法符合读写属性的委托对象的要求，这意味着 MutableMap 对象可作为读写对象的委托。

因此程序可将类只读属性委托给 Map 对象管理，在这种情况下，对象本身并不负责存储对象状态，而是将对象状态保存在 Map 集合中。这么做的好处是，当程序需要与外部接口（如 JSON）通信时，程序并不需要将该对象直接暴露出来，只要将该对象属性所委托的 Map 暴露出来即可，而 Map 则完整地保存了整个对象状态。

例如，如下程序使用 Map 存储对象的只读属性。

程序清单：codes\08\8.8\MapDelegation.kt

```
class Item(val map: Map<String, Any?>) {
    val barCode: String by map
    val name: String by map
    val price: Double by map
}
fun main(args: Array<String>) {
    val item = Item(mapOf(
        "barCode" to "133355",
        "name" to "疯狂 Kotlin 讲义",
        "price" to 68.9
    ))
    println(item.barCode)
    println(item.name)
    println(item.price)
    println("---------------")
    // 将对象持有的 map 暴露出来，其他程序可通过标准 Map 读取数据
    val map = item.map
    println(map["barCode"])
```

```
    println(map["name"])
    println(map["price"])
}
```

从上面代码可以看出，程序可以将 Item 对象的所有只读属性都委托给一个 Map 对象，其实道理很简单：Item 对象的每个属性名就相当于 Map 的 key，属性值就相当于 key 对应的 value。

指定 Map 对象作为 Item 对象的委托之后，接下来程序只要把 Item 的 Map 暴露出来，程序即可通过该 Map 来获取 Item 对象的状态，如上面程序中最后三行粗体字代码所示。

运行上面程序，可看到如下输出：

```
133355
疯狂 Kotlin 讲义
68.9
---------------
133355
疯狂 Kotlin 讲义
68.9
```

如果对象包含的属性是读写属性，则需要使用 MutableMap 作为委托对象。例如如下程序。

程序清单：codes\08\8.8\MutableMapDelegation.kt

```
class MutableItem(val map: MutableMap<String, Any?>) {
    var barCode: String by map
    var name: String by map
    var price: Double by map
}
fun main(args: Array<String>) {
    val item = MutableItem(mutableMapOf(
    ))
    // 设置 item 对象的属性，其实会委托给 MutableMap 处理
    item.barCode = "133333"
    item.name = "疯狂 Kotlin 讲义"
    item.price = 69.8
    println("---------------")
    // 将对象持有的 map 暴露出来，其他程序可通过标准 Map 读取数据
    val map = item.map
    println(map["barCode"])  // 输出 133333
    println(map["name"])  // 输出疯狂 Kotlin 讲义
    println(map["price"])  // 输出 69.8
}
```

上面程序将 MutableItem 的读写属性委托给 MutableMap 对象，这样当程序设置 MutableItem 对象的属性值时，实际上是交给 MutableMap 负责处理的，相当于为 MutableMap 设置了 key-value 对。因此程序后面可通过 MutableMap 获取到为 Item 设置的属性值。

反过来，如果程序为被委托的 MutableMap 对象设置了 key-value 对，实际上就是修改了 Item 对象的状态，如下面代码所示。

```
map["price"] = 78.8
println(item.price)  // 输出 78.8
```

8.8.6 局部属性委托

从 Kotlin 1.1 开始，Kotlin 支持为局部变量指定委托对象。这种指定了委托对象的局部变量被称为“局部委托属性”——其实还是局部变量，只是对该变量的读取、赋值操作将会交给

委托对象去实现。

与前面介绍的属性委托相似，局部变量的委托对象同样要遵守一定的要求。对于只读属性而言，同样需要实现 operator 修饰的 getValue(thisRef: Nothing?, property: KProperty<*>)方法，注意该方法的第一个参数是 Nothing?类型，Nothing 是 Kotlin 提供的一个特殊的类，专门用于代表永不存在的对象。

由于局部变量不属于任何对象，因此它的委托对象的 getValue()方法的第一个参数自然也就不存在了，因此 Kotlin 使用 Nothing?来声明 getValue()方法的第一个参数的类型。

对于读写属性而言，则需要实现 getValue()和 setValue()两个方法，其中 setValue()方法的第一个参数的类型也应该声明为 Nohting?，道理是一样的。

下面是一个局部属性委托的示例。

程序清单：codes\08\8.8\LocalDelegation.kt

```
import kotlin.reflect.*;
class MyDelegation{
    private var _backValue = "默认值"
    operator fun getValue(thisRef: Nothing?, property: KProperty<*>): String {
        println("${thisRef}的${property.name}属性执行 getter 方法")
        return _backValue
    }
    operator fun setValue(thisRef: Nothing?, property: KProperty<*>,
        newValue: String) {
        println("${thisRef}的${property.name}属性执行 setter 方法" +
            ", 传入参数值为：${newValue}")
        _backValue = newValue
    }
}
fun main(args: Array<String>) {
    var name: String by MyDelegation()
    // 访问局部变量，实际上是调用 MyDelegation 对象的 getValue()方法
    println(name)
    // 对局部变量赋值，实际上是调用 MyDelegation 对象的 setValue()方法
    name = "新的值"
    println(name)
}
```

从上面两行粗体字代码可以看出，局部变量的委托对象的 getValue()和 setValue()方法的第一个参数类型变成了 Nothing?，其他变化不大。

编译、运行上面程序，可看到如下输出：

```
null 的 name 属性执行 getter 方法
默认值
null 的 name 属性执行 setter 方法，传入参数值为：新的值
null 的 name 属性执行 getter 方法
新的值
```

从上面的运行结果也可以看出，局部变量不属于任何对象。

与普通属性类似的是，程序当然也可以利用 Kotlin 所提供的标准委托，比如可以利用 lazy()函数对局部变量延迟初始化。程序如下。

程序清单：codes\08\8.8\LocalLazy.kt

```
fun main(args: Array<String>) {
```

```
    val name: String by lazy {
        println("计算 name 局部变量")
        "疯狂 Kotlin 讲义"
    }
    // 第一次访问 name 属性时，会执行 Lambda 表达式
    println(name)
    // 以后再次访问 name 属性时，则直接使用第一次计算的值
    println(name)
}
```

上面程序中粗体字代码定义了一个延迟初始化的局部变量，当程序第一次访问该局部变量时，系统会计算 Lambda 表达式，并将表达式的值赋给该局部变量；当程序以后再次访问该局部变量时，程序将直接输出该局部变量的值，不再计算 Lambda 表达式。

运行上面程序，可看到如下输出：

```
计算 name 局部变量
疯狂 Kotlin 讲义
疯狂 Kotlin 讲义
```

8.8.7 委托工厂

除提供 getValue()、setValue()方法的对象可作为属性的委托对象之外，从 Kotlin 1.1 开始，一种类似于“委托工厂”的对象也可作为委托对象。委托工厂需要提供如下方法。

- operator fun provideDelegate(thisRef: Any?, prop: KProperty<*>)：该方法的两个参数与委托的 getValue()方法的两个参数的意义相同。

如果上面方法返回 ReadOnlyProperty 对象，那么该对象可作为只读属性的委托对象；如果返回 ReadOnlyProperty 对象，则该对象就可作为读写属性的委托对象。



提示：

Kotlin 的官方文档将这个功能称为“Providing a delegate”，说明 provideDelegate()方法负责生成委托。也就是说，包含该方法的对象相当于“委托工厂”。

使用 provideDelegate()方法来生成委托的好处是，Kotlin 会保证对象的属性被初始化时调用该方法来生成委托，这样我们即可在该方法中加入任何自定义代码，完成任何自定义逻辑。

下面示例示范了“委托工厂”的用法。

程序清单：codes\08\8.8\ProvideProperty.kt

```
import kotlin.reflect.*;
import kotlin.properties.*;
class MyTarget{
    // 该属性的委托对象是 provideDelegate()方法返回的 MyDelegation 对象
    var name: String by PropertyChecker()
}
class PropertyChecker() {
    operator fun provideDelegate(thisRef: MyTarget,
        prop: KProperty<*>): ReadWriteProperty<MyTarget, String> {
        // 插入自定义代码，可执行任意业务操作
        checkProperty(thisRef, prop.name)
        // 返回实际的委托对象
        return MyDelegation()
    }
    private fun checkProperty(thisRef: MyTarget, name: String) {
```

```
            println(" ----- 检查属性 ----");
        }
    }
    class MyDelegation: ReadWriteProperty<MyTarget, String>{
        private var _backValue = "默认值"
        override operator fun getValue(thisRef: MyTarget, property: KProperty<*>):
String {
            println("${thisRef}的${property.name}属性执行 getter 方法")
            return _backValue
        }
        override operator fun setValue(thisRef: MyTarget, property: KProperty<*>, value:
String) {
            println("${thisRef}的${property.name}属性执行 setter 方法" +
                ", 传入参数值为: ${value}")
            _backValue = value
        }
    }
    fun main(args: Array<String>) {
        // 创建对象（初始化属性），调用委托工厂的 provideDelegate()方法
        val pd = MyTarget()
        // 读取属性，实际上是调用属性的委托对象的 getter 方法
        println(pd.name)
        // 写入属性，实际上是调用属性的委托对象的 setter 方法
        pd.name = "fkit.org"
        println(pd.name)
    }
```

上面程序中第一行粗体字代码指定属性的委托对象为 PropertyChecker，该对象是一个委托工厂，它提供了 provideDelegate()方法来生成委托，该方法在生成委托之前先调用 checkProperty()方法执行检查——具体检查什么，由开发者自己决定；然后该方法才返回真正的委托对象：MyDelegation。

从上面描述可以看出，使用委托工厂的好处在于：程序可以在属性初始化时自动回调委托工厂的 provideDelegate()方法，而该方法则可以执行自己的业务操作。

编译、运行上面程序，可看到如下输出：

```
----- 检查属性 ----
MyTarget@41906a77 的 name 属性执行 getter 方法
默认值
MyTarget@41906a77 的 name 属性执行 setter 方法，传入参数值为：fkit.org
MyTarget@41906a77 的 name 属性执行 getter 方法
fkit.org
```

从上面的运行结果来看，属性的委托依然由 MyDelegation 对象负责，只是在属性初始化时插入了业务检查方法：checkProperty()。

8.9 本章小结

本章主要介绍了 Kotlin 面向对象的深入部分。本章详细介绍了 Kotlin 提供的扩展机制，这种扩展机制允许开发者为已有的类动态添加属性和方法，这种机制使用起来非常便捷。读者不仅需要掌握扩展的语法和用法，还需要理解 Kotlin 关于扩展的实现机制，以便在 Java 代码中也可调用 Kotlin 的扩展成员。

本章还介绍了 Kotlin 的 const 常量，使用 final 修饰符修饰变量、方法和类的用法，讲解了

抽象类和接口的用法，并深入比较了接口和抽象类之间的联系与区别，以便读者能掌握接口和抽象类在用法上的区别。

本章介绍了嵌套类、内部类的概念和用法，笔者从 Kotlin 与 Java 对比的角度进行了讲解，让读者明白 Kotlin 的嵌套类就是 Java 的静态内部类，Kotlin 的内部类就是 Java 的非静态内部类，这样以便读者能迅速抓住 Kotlin 知识的要点。Kotlin 使用强大的对象表达式代替了原来的匿名内部类，因此对象表达式相当于增强版的匿名内部类。

本章介绍了 Kotlin 的对象声明。对象声明是一种语法级别的单例模式，只要读者用对象声明来创建对象，即可创建单例对象。

伴生对象是 Kotlin 的一个新功能，也是读者需要认真学习的。但这个功能是 Java 不需要的。Java 本身有 static 关键字，因此可定义静态成员；但 Kotlin 没有 static 关键字，因此 Kotlin 需要使用伴生对象来模拟静态成员。本章也介绍了 Kotlin 的枚举类。Kotlin 的枚举类与 Java 的枚举类差别不大，读者应该不难掌握。

本章的另一个重点内容是委托，包括类委托和属性委托。尤其是属性委托，它对一些现代编程语言的功能提供了支持，比如延迟属性、属性监听、使用 Map 存储属性等，通过属性委托，我们可以对属性的操作进行集中管理，这种机制非常有吸引力。

CHAPTER

9

第9章 异常处理

本章要点

- 使用 try...catch 捕获异常
- 异常类的继承体系
- 异常对象的常用属性和方法
- finally 块的作用
- 异常处理的合理嵌套
- 使用 throw 抛出异常
- 自定义异常
- 异常链和异常转译
- 异常的跟踪栈信息

Kotlin 同样提供了完善的异常处理机制，异常处理机制可以使程序中的异常处理代码和正常业务代码分离，保证程序代码更加优雅，并可以提高程序的健壮性。

Kotlin 的异常处理机制主要依赖于 try、catch、finally、throw 这四个关键字，其中 try 关键字后紧跟一个用花括号括起来的代码块（花括号不可省略），简称 try 块，它里面放置可能引发异常的代码。catch 后对应异常类型和一个代码块，表明该 catch 块用于处理这种类型的代码块。多个 catch 块后还可以跟一个 finally 块，用于回收在 try 块里打开的物理资源，异常处理机制会保证 finally 块总被执行。而 throw 用于抛出一个实际的异常，throw 可以单独作为语句使用，抛出一个具体的异常对象。

与 Java 的异常处理机制相比，Kotlin 抛弃了 checked 异常，相当于所有异常都是 runtime 异常，这意味着开发者想捕获异常就捕获，不想捕获异常也行，不需要使用 throws 关键字声明抛出异常。

9.1 异常处理机制

Kotlin 的异常处理机制可以让程序具有极好的容错性，让程序更加健壮。当程序运行出现意外情形时，系统会自动生成一个 Exception 对象来通知程序，从而实现将“业务功能实现代码”和“错误处理代码”分离，提供了更好的可读性。

9.1.1 使用 try...catch 捕获异常

Kotlin 的异常处理机制的语法结构如下：

```
try {
    // 业务实现代码
    ...
} catch (e: Exception) {
    // 异常处理代码
    ...
}
...
finally {
    // 可选的 finally 块
}
```

与 Java 的异常处理流程相同，整个异常处理流程可包含 1 个 try 块、0~*N* 个 catch 块、0~1 个 finally 块，但 catch 块与 finally 块至少出现其中之一。

如果在执行 try 块中的业务逻辑代码时出现异常，系统将自动生成一个异常对象，该异常对象被提交给运行时环境，这个过程被称为抛出（throw）异常。

当运行时环境收到异常对象时，会寻找能处理该异常对象的 catch 块，如果找到了合适的 catch 块，就把该异常对象交给该 catch 块处理，这个过程被称为捕获（catch）异常；如果运行时环境找不到捕获异常的 catch 块，则运行时环境中止，程序也将退出。



提示：

不管程序代码块是否处于 try 块中，甚至包括 catch 块中的代码，只要执行该代码块时出现了异常，系统总会自动生成一个异常对象。如果程序没有为这段代码定义任何的 catch 块，则运行时环境无法找到处理该异常的 catch 块，程序就在此退出。

下面使用异常处理机制来改写本书第5章的五子棋游戏中用户下棋部分的代码。

程序清单：codes\09\9.1\ExceptionTest.kt

```
import java.io.*
fun main(args: Array<String>) {
    try {
        val fis = FileInputStream("a.txt")
        println(fis.read())
    } catch(e: Exception) {
        println("读取文件出现异常")
    }
}
```

上面程序把处理用户输入字符串的代码都放在try块中，只要程序出现异常，系统就将抛出一个异常对象，并把这个异常对象交给对应的catch块（也就是上面程序中的粗体字代码）处理，catch块的处理方式是向用户提示“读取文件出现异常”。

有些时候，程序在try块中打开了一些物理资源（例如数据库连接、网络连接和磁盘文件等），这些物理资源都必须显式回收。

在哪里回收这些物理资源呢？在try块中回收，还是在catch块中回收呢？假设程序在try块中进行资源回收，如果try块中的某条语句引起了异常，该语句后的其他语句通常不会获得执行的机会，这将导致位于该语句之后的资源回收语句得不到执行。如果在catch块中进行资源回收，但catch块完全有可能得不到执行，这将导致不能及时回收这些物理资源。

为了保证一定能回收try块中打开的物理资源，异常处理机制提供了finally块。不管try块中的代码是否出现异常，也不管哪一个catch块被执行，甚至在try块或catch块中执行了return语句，finally块总会被执行。看如下程序。

程序清单：codes\09\9.1\FinallyTest.kt

```
import java.io.*
fun main(args: Array<String>) {
    var fis: FileInputStream? = null
    try {
        fis = FileInputStream("a.txt")
    } catch (ioe: IOException) {
        println(ioe.message)
        // return语句强制方法返回
        return      // ①
        // 使用exit退出虚拟机
        // System.exit(1)     // ②
    } finally {
        // 关闭磁盘文件，回收资源
        fis?.close()
        println("执行finally块里的资源回收!")
    }
}
```

上面程序在try块后增加了finally块，用于回收在try块中打开的物理资源。注意程序的catch块中①处有一条return语句，该语句强制方法返回。在通常情况下，一旦在方法中执行到return语句的地方，程序将立即结束该方法；而现在不会了，虽然return语句也强制方法结束，但一定会先执行finally块中的代码。运行上面程序，可看到如下结果：

```
a.txt (系统找不到指定的文件。)
执行finally块里的资源回收!
```

上面的运行结果表明在方法返回之前还是执行了 finally 块中的代码。将①处的 return 语句注释掉，取消②处代码的注释，即在异常处理的 catch 块中使用 System.exit(1)语句来退出虚拟机。运行上面代码，可看到如下结果：

```
a.txt (系统找不到指定的文件。)
```

上面的运行结果表明 finally 块没有被执行。如果在异常处理代码中使用 System.exit(1)语句来退出虚拟机，则 finally 块将失去执行的机会。

注意：

除非在 try 块、catch 块中调用了退出虚拟机的方法，否则不管在 try 块、catch 块中执行怎样的代码，出现怎样的情况，异常处理的 finally 块总会被执行。



在通常情况下，不要在 finally 块中使用如 return 或 throw 等导致方法中止的语句，（throw 语句将在后面介绍），一旦在 finally 块中使用了 return 或 throw 语句，就将会导致 try 块、catch 块中的 return、throw 语句失效。看如下程序。

程序清单：codes\09\9.1\FinallyFlowTest.kt

```
fun main(args: Array<String>) {
    var a = test()
    println(a)
}
fun test(): Boolean {
    try {
        // 因为 finally 块中包含了 return 语句
        // 所以下面的 return 语句失去作用
        return true
    } finally {
        return false
    }
}
```

上面程序在 finally 块中定义了一条 return false 语句，这将导致 try 块中的 return true 语句失去作用。运行上面程序，将打印出 false 结果。

当程序执行 try 块、catch 块时遇到了 return 或 throw 语句，这两条语句都会导致该方法立即结束，但是系统执行这两条语句并不会结束该方法，而是去寻找该异常处理流程中是否包含 finally 块，如果没有 finally 块，程序将立即执行 return 或 throw 语句，方法中止；如果有 finally 块，系统就立即开始执行 finally 块——只有当 finally 块执行完成后，系统才会再次跳回来执行 try 块、catch 块中的 return 或 throw 语句；如果 finally 块中也使用了 return 或 throw 等导致方法中止的语句，finally 块已经中止了方法，系统将不会跳回去执行 try 块、catch 块中的任何代码。

注意：

尽量避免在 finally 块中使用 return 或 throw 等导致方法中止的语句，否则可能出现一些很奇怪的情况。



9.1.2 异常类的继承体系

与 Java 的异常处理流程类似，当运行时环境接收到异常对象后，会依次判断该异常对象是否是 catch 块后异常类或其子类的实例，如果是，运行时环境将调用该 catch 块来处理该异常；否则，将再次拿该异常对象和下一个 catch 块中的异常类进行比较。Kotlin 的异常处理流程示意图如图 9.1 所示。

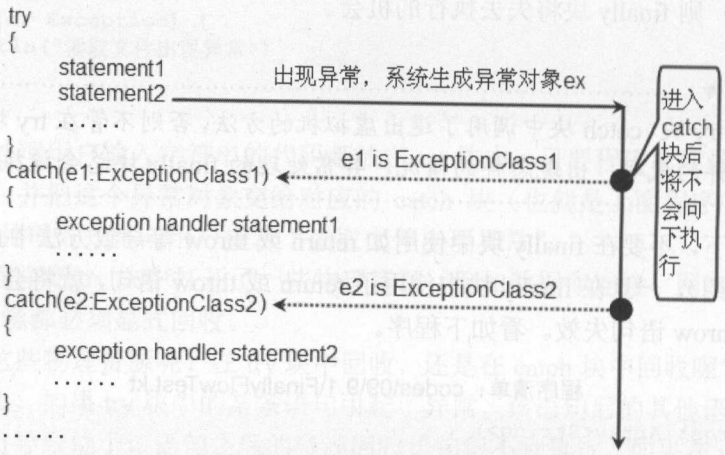


图 9.1 Kotlin 的异常处理流程示意图

当程序进入负责异常处理的 catch 块时，系统生成的异常对象 ex 将会传给 catch 块后的异常形参，从而允许 catch 块通过该对象来获得异常的详细信息。

从图 9.1 中可以看出，在 try 块后可以有多个 catch 块，这是为了针对不同的异常类提供不同的异常处理方式。当系统发生不同的意外情况时，系统会生成不同的异常对象，运行时环境就会根据该异常对象所属的异常类来决定使用哪个 catch 块来处理该异常。

通过在 try 块后提供多个 catch 块可以无须在异常处理块中使用 if、switch 来判断异常类型，但依然可以针对不同的异常类型提供相应的处理逻辑，从而提供更细致、更有条理的异常处理逻辑。

从图 9.1 中可以看出，在通常情况下，如果 try 块被执行一次，则在 try 块后只有一个 catch 块会被执行，绝不可能有多个 catch 块被执行。除非在循环中使用了 continue 语句开始下一次循环，下一次循环又重新运行了 try 块，这才可能导致多个 catch 块被执行。

注意：

try 块与 if 语句不一样，try 块后的花括号（{...}）不可以省略，即使 try 块中只有一行代码，也不可省略这对花括号。与之类似的是，catch 块后的花括号（{...}）也不可以省略。还有一点需要指出：在 try 块中声明的变量是代码块内的局部变量，它只在 try 块内有效，在 catch 块中不能访问该变量。

Kotlin 提供了 kotlin.Throwable 类，所有的异常类都是 Throwable 类的子类。此外，Kotlin 还通过类型别名的方式引入了 Java 的 Error 和 Exception 两个子类，但 Kotlin 并没有严格区分错误和异常（在 Java 中 Error 和 Exception 的区别也不明显）。此外，Kotlin 完全借用了 Java 的异常体系，这些异常类之间有严格的继承关系。

下面看一个异常捕获的例子。

程序清单：codes\09\9.1\DivTest.kt

```
fun main(args: Array<String>) {
    try{
        var a = Integer.parseInt(args[0])
        var b = Integer.parseInt(args[1])
        val c = a / b
        println("您输入的两个数相除的结果是：${c}")
    } catch (ie: IndexOutOfBoundsException) {
        println("数组越界：运行程序时输入的参数个数不够")
    } catch (ne: NumberFormatException) {
        println("数字格式异常：程序只能接收整数参数")
    } catch (ae: ArithmeticException) {
        println("算术异常")
    } catch (e: Exception) {
        println("未知异常")
    }
}
```

上面程序针对 IndexOutOfBoundsException、NumberFormatException、ArithmeticException 类型的异常提供了专门的异常处理逻辑。程序运行时的异常处理逻辑可能有如下几种情形。

- 如果运行该程序时输入的参数不够，将会发生数组越界异常，运行时环境将调用 IndexOutOfBoundsException 对应的 catch 块处理该异常。
- 如果运行该程序时输入的参数不是数字，而是字母，将发生数字格式异常，运行时环境将调用 NumberFormatException 对应的 catch 块处理该异常。
- 如果运行该程序时输入的第二个参数是 0，将发生除 0 异常，运行时环境将调用 ArithmeticException 对应的 catch 块处理该异常。
- 如果程序运行时出现其他异常，该异常对象总是 Exception 类或其子类的实例，运行时环境将调用 Exception 对应的 catch 块处理该异常。

上面程序中的 3 种异常，都是非常常见的运行时异常，读者应该记住这些异常，并掌握在哪些情况下可能出现这些异常。

正如在前面程序中所看到的，程序总是把对应 Exception 类的 catch 块放在最后，这是为什么呢？想一下图 9.1 所示的异常处理流程，读者可能就明白原因了：如果把 Exception 类对应的 catch 块排在其他 catch 块的前面，Kotlin 运行时将直接进入该 catch 块（因为所有的异常对象都是 Exception 或其子类的实例），而排在它后面的 catch 块将永远也不会获得执行的机会。

实际上，在进行异常捕获时不仅应该把 Exception 类对应的 catch 块放在最后，而且应该把用来捕获所有父类异常的 catch 块都排在用来捕获子类异常的 catch 块的后面（简称：先处理小异常，再处理大异常）；否则将出现编译错误。看如下代码片段：

```
try{
    statements...
    println(a)
} catch( e: RuntimeException) {     // ①
    println("运行时异常")
} catch ( ne: NullPointerException) {    // ②
    println("空指针异常")
}
```

上面代码中有两个 catch 块，前一个 catch 块捕获 RuntimeException 异常，后一个 catch 块

捕获 NullPointerException 异常。编译上面代码，将会在②处出现已捕获到异常 java.lang.NullPointerException 的错误，因为①处的 RuntimeException 已经包括了 NullPointerException 异常，所以②处的 catch 块永远也不会获得执行的机会。

注意：

在进行异常捕获时，一定要记住先捕获小异常，再捕获大异常。在这一点上，Java 比 Kotlin 更出色，对于上面的代码，Java 编译器会在②处报错：异常已捕获；而 Kotlin 编译器不会报错，但这样的代码不应该出现在程序中。



9.1.3 访问异常信息

如果程序需要在 catch 块中访问异常对象的相关信息，则可以通过访问 catch 块后的异常形参来获得。当运行时环境决定调用某个 catch 块来处理该异常对象时，会将异常对象赋给 catch 块后的异常参数，程序即可通过该参数来获得异常的相关信息。

所有的异常对象都包含了如下几个常用属性和方法。

- message：该属性返回该异常的详细描述字符串。
- stackTrace：该属性返回该异常的跟踪栈信息。
- printStackTrace()：将该异常的跟踪栈信息输出到标准错误输出。
- printStackTrace(PrintStream s)：将该异常的跟踪栈信息输出到指定输出流。

下面例子程序演示了如何访问异常信息。

程序清单：codes\09\9.1\AccessExceptionMsg.kt

```
import java.io.*
fun main(args: Array<String>) {
    try{
        var fis = FileInputStream("a.txt")
    } catch (ioe: IOException) {
        println(ioe.message)
        ioe.printStackTrace()
    }
}
```

上面程序调用了 Exception 对象的 message 属性来获得异常对象的详细信息，也使用了 printStackTrace()方法来打印该异常的跟踪栈信息。运行上面程序，将会看到如图 9.2 所示的界面。

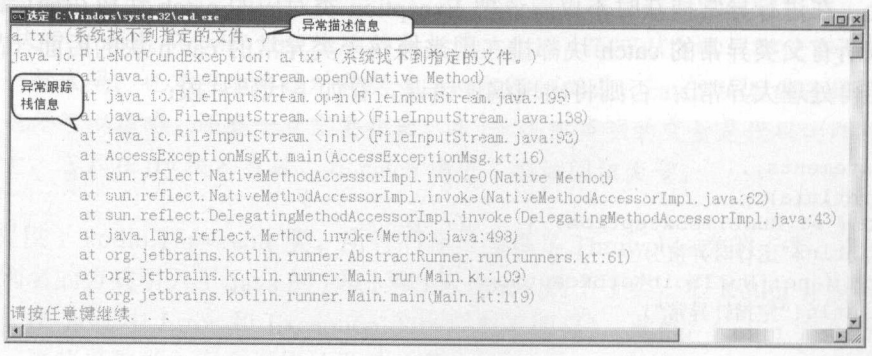


图 9.2 访问异常信息

从图 9.2 中可以看到异常的详细描述信息："a.txt (系统找不到指定的文件)"，这就是调用异常的 message 属性返回的字符串。下面更详细的信息是该异常的跟踪栈信息。关于异常的跟踪栈信息后面还有更详细的介绍，此处不再赘述。

9.1.4 异常处理嵌套

异常处理流程代码可以放在任何能放可执行代码的地方，因此完整的异常处理流程既可放在 try 块中，也可放在 catch 块中，还可放在 finally 块中，这样就会形成异常嵌套。

异常处理嵌套的深度没有很明确的限制，但通常没有必要使用超过两层的嵌套异常处理，因为嵌套层次太深的话会导致程序可读性降低。

9.1.5 try 语句是表达式

与 if 语句类似，Kotlin 的 try 语句也是表达式，因此 try 语句也可用于对变量赋值。try 表达式的返回值是 try 块中的最后一个表达式的值，或者是被执行的 catch 块中的最后一个表达式的值，finally 块中的内容不会影响表达式的结果。

如下程序示范了 try 语句是一个表达式的效果。

程序清单：codes\09\9.1\TryExpr.kt

```
fun main(args: Array<String>) {
    val input = readLine()
    // 用 try 表达式对变量 a 赋值
    val a: Int? = try {Integer.parseInt(input) } catch (e: NumberFormatException)
{ null }
    println(a)
}
```

上面的粗体字代码用 try 表达式对变量 a 赋值，如果用户输入的 input 是合法的整数字符串，程序即可将它转换为整数；如果用户输入的 input 不是合法的整数字符串，运行时环境进入执行 catch 块，因此该 try 表达式为 null。

9.2 使用 throw 抛出异常

与 Java 类似，Kotlin 也允许程序自行抛出异常，自行抛出异常使用 throw 语句来完成。

9.2.1 抛出异常

如果需要在程序中自行抛出异常，则应使用 throw 语句。throw 语句可以单独使用，throw 语句抛出的不是异常类，而是一个异常实例，而且每次只能抛出一个异常实例。throw 语句的语法格式如下：

```
throw ExceptionInstance
```

由于 Kotlin 没有 checked 异常（即使某个异常在 Java 中原本是 checked 异常，在 Kotlin 中它也不是 checked 异常），因此 Kotlin 抛出异常的语句无须放在 try 块中，程序既可以显式使用 try...catch 来捕获并处理该异常，也可以完全不理会该异常，把该异常交给该方法调用者处理。例如下面例子程序。

程序清单：codes\09\9.2\ThrowTest.kt

```
fun main(args: Array<String>) {
    // 无论该异常在 Java 中是否为 checked 异常
    // 在 Kotlin 中该异常都不是 checked 异常
    throwChecked(-3)
    throwRuntime(3)
}
fun throwChecked(a: Int) {
    if (a > 0) {
        // 自行抛出普通异常，在 Kotlin 中也不是 checked 异常
        // 该代码不必处于 try 块中
        throw Exception("a 的值大于 0，不符合要求")
    }
}
fun throwRuntime(a: Int) {
    if (a > 0) {
        throw RuntimeException("a 的值大于 0，不符合要求")
    }
}
```

上面程序中的两行粗体字代码分别抛出了 Exception 和 RuntimeException 异常，Exception 在 Java 中就是 checked 异常，但在 Kotlin 中不是，所以我们看到上面两个方法对两种不同异常的处理完全相同。

9.2.2 自定义异常类

在通常情况下，程序很少会自行抛出系统异常，因为异常类的类名通常也包含了该异常的有用信息。所以在选择抛出异常时，应该选择合适的异常类，从而可以明确地描述该异常情况。在这种情形下，应用程序常常需要抛出自定义异常。

用户自定义异常都应该继承 Exception 基类，定义异常类时通常需要提供两个构造器：一个是无参数的构造器；另一个是带一个字符串参数的构造器，这个字符串将作为该异常对象的描述信息（也就是异常对象的 message 属性的返回值）。

下面例子程序创建了一个自定义异常类。

程序清单：codes\09\9.2\AuctionException.kt

```
class AuctionException: Exception{
    // 无参数的构造器
    constructor() {}        // ①
    // 带一个字符串参数的构造器
    constructor(msg: String): super(msg) {}  // ②
}
```

上面程序创建了 AuctionException 异常类，并为该异常类提供了两个构造器。尤其是②号粗体字代码创建的带一个字符串参数的构造器，该构造器通过 super 委托调用父类的构造器，正是这个 super 调用可以将此字符串参数传给异常对象的 message 属性，该 message 属性就是该异常对象的详细描述信息。

在大部分情况下，创建自定义异常都可采用与 AuctionException.kt 相似的代码完成，只需改变 AuctionException 异常类的类名即可，让该异常类的类名可以准确描述该异常。

9.2.3 catch 和 throw 同时使用

实际应用的异常处理可能需要更复杂的处理方式——当一个异常出现时，单靠某个方法无法完全处理该异常，必须通过几个方法协作才可完全处理该异常。也就是说，在异常出现的当前方法中，程序只对异常进行部分处理，还有些处理需要在该方法的调用者中才能完成，所以应该再次抛出异常，让该方法的调用者也能捕获到异常。

为了实现这种通过多个方法协作处理同一个异常的情形，可以在 catch 块中结合 throw 语句来完成。如下例子程序示范了这种 catch 和 throw 同时使用的方法。

程序清单：codes\09\9.2\AuctionTest.kt

```
public class AuctionTest{
    var initPrice: Double = 30.0
    fun bid(bidPrice: String){
        var d: Double
        try {
            d = bidPrice.toDouble()
        } catch (e: Exception) {
            // 此处完成本方法中可以对异常执行的修复处理
            // 此处仅仅是在控制台打印异常的跟踪栈信息
            e.printStackTrace()
            // 再次抛出自定义异常
            throw AuctionException("竞拍价必须是数值，"
                + "不能包含其他字符！")
        }
        if (initPrice > d){
            throw AuctionException("竞拍价比起拍价低，"
                + "不允许竞拍！")
        }
        initPrice = d
    }
}
fun main(args: Array<String>) {
    val at = AuctionTest()
    try{
        at.bid("df")
    }catch (ae: AuctionException){
        // 再次捕获到bid()方法中的异常，并对该异常进行处理
        println(ae.message)
    }
}
```

上面程序中粗体字代码对应的 catch 块捕获到异常后，系统打印了该异常的跟踪栈信息，接着抛出一个 AuctionException 异常，通知该方法的调用者再次处理该 AuctionException 异常。所以程序中的 main()方法，也就是 bid()方法调用者还可以再次捕获 AuctionException 异常，并将该异常的详细描述信息输出到标准输出。

这种 catch 和 throw 结合使用的情况在大型企业级应用中非常常用。企业级应用对异常的处理通常分成两个部分：① 应用后台需要通过日志来记录异常发生的详细情况；② 应用还需要根据异常向应用使用者传达某种提示。在这种情形下，所有异常都需要两个方法共同完成，也就必须结合使用 catch 和 throw。

9.2.4 异常链

对于真实的企业级应用而言，常常有严格的分层关系，层与层之间有非常清晰的划分，上层功能的实现严格依赖于下层的 API，也不会跨层访问。图 9.3 显示了这种具有分层结构应用的大致示意图。

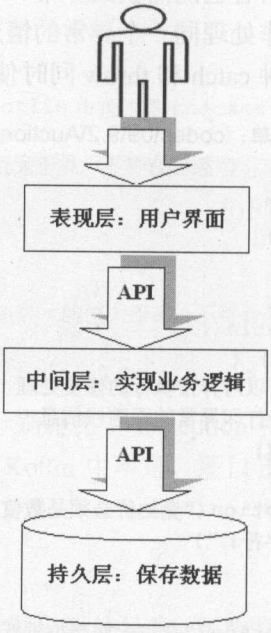


图 9.3 分层结构示意图

对于一个采用图 9.3 所示结构的应用来说，当业务逻辑层访问持久层出现 SQLException 异常时，程序不应该把底层的 SQLException 异常传到用户界面，有如下两个原因。

- 对于正常用户而言，他们不想看到底层的 SQLException 异常，SQLException 异常对他们使用该系统没有任何帮助。
- 对于恶意用户而言，将 SQLException 异常暴露出来不安全。

把底层的原始异常直接传给用户是一种不负责任的表现。通常的做法是：程序先捕获原始异常，然后抛出一个新的业务异常，新的业务异常中包含了对用户的提示信息。这种处理方式被称为异常转译。假设程序需要实现计算工资的方法，则应该采用如下结构的代码来实现该方法。

```
fun calSal(){
    try {
        // 实现计算工资的业务逻辑
        ...
    } catch(sqle: SQLException) {
        // 把原始异常记录下来，留给管理员
        ...
        // 下面异常中的 message 就是对用户的提示
        throw SalException("访问底层数据库出现异常")
    } catch(e: Exception) {
        // 把原始异常记录下来，留给管理员
        ...
        // 下面异常中的 message 就是对用户的提示
```

```
            throw SalException("系统出现未知异常")
        }
    }
```

这种把原始异常信息隐藏起来，仅向上提供必要的异常提示信息的处理方式，可以保证底层异常不会扩散到表现层，可以避免向上暴露太多的实现细节，这完全符合面向对象的封装原则。

捕获一个异常，然后抛出另一个异常，并把原始异常信息保存下来，这是一种典型的链式处理（23 种设计模式之一：职责链模式），也被称为“异常链”。

Kotlin 的 Throwable 类及其子类在构造器中都可以接收一个 cause 对象作为参数。这个 cause 就用来表示原始异常，这样可以把原始异常传递给新的异常，使得即使在当前位置创建并抛出了新的异常，我们也能通过这个异常链追踪到异常最初发生的位置。例如，希望通过上面的 SalException 跟踪到最原始的异常信息，则可以将该方法改写为如下形式：

```
fun calSal() {
    try {
        // 实现计算工资的业务逻辑
        ...
    } catch(sqle: SQLException) {
        // 把原始异常记录下来，留给管理员
        ...
        // 下面异常中的 sqle 就是原始异常
        throw SalException(sqle)
    } catch(e: Exception) {
        // 把原始异常记录下来，留给管理员
        ...
        // 下面异常中的 e 就是原始异常
        throw SalException(e)
    }
}
```

上面程序中粗体字代码在创建 SalException 对象时，传入了一个 Exception 对象，而不是传入了一个 String 对象，这就需要 SalException 类有相应的构造器。在 JDK 1.4 以后，Throwable 基类已经有了一个可以接收 Exception 参数的方法，所以可以采用如下代码来定义 SalException 类。

程序清单：codes\09\9.2\SalException.kt

```
class SalException: Exception {
    constructor(){}
    constructor(msg: String): super(msg) {}
    // 创建一个可以接收 Throwable 参数的构造器
    constructor(t: Throwable): super(t) {}
}
```

创建了这个 SalException 业务异常类后，就可以用它来封装原始异常，从而实现对异常的链式处理。

9.2.5 throw 语句是表达式

与 try 语句是表达式一样，Kotlin 的 throw 语句也是表达式，但由于 throw 表达式的类型比较特殊，是 Nothing 类型，因此很少将 throw 语句赋值给其他变量，但我们可以在 Elvis 表达式中使用 throw 表达式。例如如下代码。

程序清单：codes\09\9.2\ThrowExpr.kt

```
class User(var name: String? = null, var pass: String? = null)
fun main(args: Array<String>) {
    val user = User()
    // 在 Elvis 表达式中使用 throw 表达式
    // throw 表达式表示程序出现异常，不会真正对变量赋值
    val th: String = user.name ?: throw NullPointerException("目标对象不能为 null")
    println(th)
}
```

上面程序中粗体字代码将 user.name 赋值给 th 变量，由于 user.name 是可空类型，因此程序对 user.name 使用了 Elvis 表达式进行判断：当 user.name 不为 null 时，将其值赋给 user.name；否则，使用 throw 表达式的值，throw 表达式的类型是 Nothing，用于表示程序无法“真正”得到该表达式的值。因此，如果 user.name 为 null，程序将会出现异常，不会对 th 变量赋值，故可将 th 变量声明为 String 类型。

编译、运行该程序，可看到程序因为 NullPointerException 异常而结束。

Nothing 类型没有值，而是用于标记永远无法“真正”返回的表达式，因此程序不会获取表达式的值。当我们自己定义函数时，也可使用 Nothing 来标记一个永远不会返回的函数（程序清单同上）。

```
fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

接下来，也可在 Elvis 表达式中使用上面的函数（程序清单同上）。

```
val s: String = user.name ?: fail("目标对象不能为 null")
println(s)
```

9.3 异常的跟踪栈

异常对象的 printStackTrace()方法用于打印异常的跟踪栈信息，根据 printStackTrace()方法的输出结果，开发者可以找到异常的源头，并跟踪到异常一路触发的过程。

看下面用于测试 printStackTrace 的例子程序。

程序清单：codes\09\9.3\PrintStackTraceTest.kt

```
class SelfException: Exception{
    constructor() {}
    constructor(msg: String): super(msg) {}
}
class PrintStackTraceTest{
    fun firstMethod(){
        secondMethod()
    }
    fun secondMethod(){
        thirdMethod()
    }
    fun thirdMethod(){
        throw SelfException("自定义异常信息")
    }
}
fun main(args: Array<String>) {
```

```
    PrintStackTraceTest().firstMethod()
}
```

上面程序中的 main()方法调用了 firstMethod()，firstMethod()调用了 secondMethod()，secondMethod()调用了 thirdMethod()，thirdMethod()则直接抛出一个 SelfException 异常。运行上面程序，将会看到如图 9.4 所示的结果。

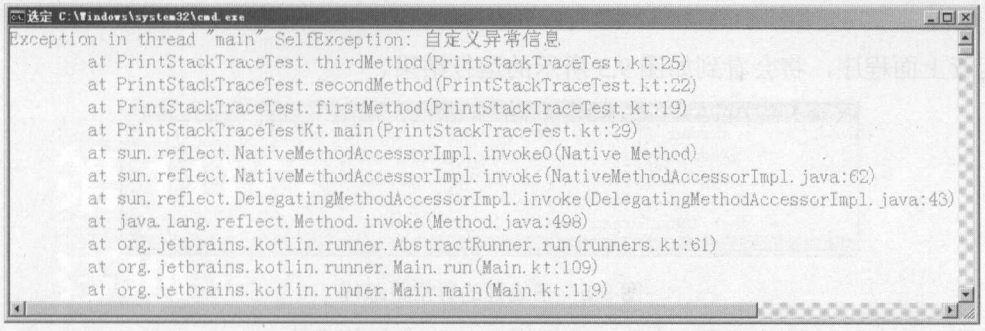


图 9.4 异常跟踪栈信息

从图 9.4 中可以看出，异常从 thirdMethod()方法开始触发，传到 secondMethod()方法，再传到 firstMethod()方法，最后传到 main()方法，由 main()方法传给 Kotlin 执行环境，这个过程就是 Kotlin 的异常跟踪栈。

在面向对象的编程中，大多数复杂操作都会被分解成一系列方法调用。这是因为：为了实现更好的可重用性，将每个可重用的代码单元定义成方法，将复杂任务逐渐分解为更易管理的小型子任务。由于一个大的业务功能需要由多个对象来共同实现，在最终的编程模型中，很多对象将通过一系列方法调用来实现通信，执行任务。

所以，面向对象的应用程序运行时，经常会发生一系列方法调用，从而形成“方法调用栈”。而异常的传播则相反：只要异常没有被完全捕获（包括异常没有被捕获，或异常被处理后重新抛出了新异常），异常就从发生异常的方法逐渐向外传播，首先传给该方法的调用者，该方法的调用者再次传给其调用者……直至最后传到 main()方法，如果 main()方法依然没有处理该异常，该异常就会传给 Kotlin 运行时环境，Kotlin 运行时环境会中止该程序，并打印异常的跟踪栈信息。如图 9.4 所示的异常跟踪栈信息非常清晰地记录了应用程序中执行停止的各个点。

第一行的信息显示了异常的类型和异常的详细消息。

接下来跟踪栈记录了程序中所有的异常发生点，各行显示了被调用方法中执行停止的位置，并标明了类、类中的方法名、与故障点对应的文件的行。一行行地往下看，跟踪栈总是最内部的被调用方法逐渐上传，直到最外部的业务操作的起点，通常就是程序的入口 main()方法或 Thread 类的 run()方法（多线程的情形）。

下面例子程序示范了多线程程序中发生异常的情形。

程序清单：codes\09\9.3\ThreadExceptionTest.kt

```
class ThreadExceptionTest: Runnable{
    override fun run(){
        firstMethod()
    }
    fun firstMethod(){
        secondMethod()
    }
    fun secondMethod(){
```

```
        val a = 5
        val b = 0
        val c = a / b
    }
}
fun main(args: Array<String>) {
    Thread(ThreadExceptionTest()).start()
}
```

运行上面程序，将会看到如图 9.5 所示的运行结果。

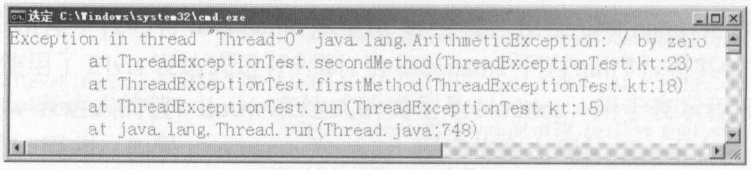


图 9.5 子线程的异常跟踪栈

从图 9.5 中可以看出，程序在 Thread 的 run()方法中出现了 ArithmeticException 异常，这个异常的源头是 ThreadExcetpionTest 的 secondMethod()方法，位于 ThreadExcetpionTest.kt 文件的第 23 行。这个异常传播到 Thread 类的 run 方法就会结束（如果该异常没有得到处理，将会导致该线程中止运行）。

前面已经讲过，调用 Exception 的 printStackTrace()方法就是打印该异常的跟踪栈信息，也就会看到如图 9.4、图 9.5 所示的信息。当然，如果方法调用的层次很深，将会看到更加复杂的异常跟踪栈。



提示：

虽然 printStackTrace()方法可以很方便地用于跟踪异常的发生情况，可以用它来调试程序，但在最后发布的程序中，应该避免使用它；而应该对捕获的异常进行适当的处理，而不是简单地将异常的跟踪栈信息打印出来。

9.4 本章小结

本章主要介绍了 Kotlin 异常处理机制的相关知识。Kotlin 的异常处理主要依赖于 try、catch、finally 和 throw 4 个关键字，本章详细讲解了这 4 个关键字的用法。本章还介绍了异常类之间的继承关系，以及在实际开发中最常用的异常链和异常转译。

CHAPTER

10

第10章 泛型

本章要点

- 定义泛型接口、泛型类
- 派生泛型接口、泛型类的子类、实现类
- 泛型型变的必要性
- 泛型型变的两种方式：协变和逆变
- Kotlin 的声明处型变
- Kotlin 的使用处型变和 Java 的泛型通配符
- Kotlin 的星号投影和 Java 的原始类型
- Kotlin 的泛型函数和 Java 的泛型方法
- 在内联函数中定义具体化的类型参数
- 设定类型形参的上限

与 Java 泛型相同，Kotlin 同样提供了泛型支持。对于简单的泛型类、泛型函数的定义，Kotlin 与 Java 的差别不大，因此读者应该可以很快掌握 Kotlin 的泛型。

Kotlin 泛型的特色功能是型变支持，Kotlin 提供了声明处型变和使用处型变两种支持，而 Java 只支持使用处型变，因此读者需要花时间理解 Kotlin 的声明处型变的功能和优势；Kotlin 的使用处型变与 Java 的使用处型变有大致的对应关系，只是在写法上存在一定的差异。

本章将会详细介绍 Kotlin 的泛型知识，包括定义泛型类、泛型接口、Kotlin 的声明处型变、使用处型变、星号投影、泛型函数，以及为类型形参设置上限等内容。

10.1 泛型入门

与 Java 类似，Kotlin 的泛型也允许在定义类、接口、函数时使用泛型形参，这个泛型形参将在声明变量、创建对象、调用方法时动态地指定（即传入实际的类型，也可称为类型实参）。

10.1.1 定义泛型接口、类

可以为任何类、接口增加泛型声明。下面自定义一个 Apple 类，这个 Apple 类就可以包含一个泛型声明。

程序清单：codes\10\10.1\Apple.kt

```
// 定义 Apple 类时使用了泛型声明
open class Apple<T>{
    // 使用泛型 T 定义属性
    open var info: T?
    constructor(){
        info = null
    }
    // 下面方法中使用泛型 T 来定义构造器
    constructor(info: T){
        this.info = info
    }
}
fun main(args: Array<String>) {
    // 由于传给泛型 T 的是 String，所以构造器的参数只能是 String
    var a1: Apple<String> = Apple<String>("苹果")
    println(a1.info)
    // 由于传给泛型 T 的是 Int，所以构造器的参数只能是 Int
    var a2: Apple<Int> = Apple(3)
    System.out.println(a2.info)
    // 由于构造器的参数是 Double，因此系统可推断出泛型形参为 Double 类型
    var a3 = Apple(5.67)
    System.out.println(a3.info)
}
```

上面程序定义了一个带泛型声明的 Apple<T>类（不要理会这个泛型形参是否具有实际意义），使用 Apple<T>类时就可为泛型 T 传入实际的类型，这样就可以生成如 Apple<String>、Apple<Int>、Apple<Double>等形式的多个逻辑子类（物理上并不存在）。

为类型定义了泛型形参之后，程序在使用该类型时即可为之传入实际的类型，这个实际的类型既可显式传入（如上面 main()函数中第一行粗体字代码所示）；也可在声明变量时指定，这样系统可在创建对象时推断出该类型参数的实际类型（如上面 main()函数中第二行粗体字代

码所示）；抑或完全不指定类型参数的实际类型，而是由系统根据参数进行推断（如上面 main() 函数中第三行粗体字代码所示）。

注意：

Kotlin 不需要使用形如 Java 的“菱形语法”，比如上面程序中第二行粗体字代码，在 Java 中必须写成 Apple<Int> a2 = new Apple<>(3)。也就是说，在 Java 中即使系统可推断出泛型参数的类型，Java 也要求至少保留两个尖括号，但这在 Kotlin 中完全不需要，直接写成 var a2: Apple<Int> = Apple(3)就行了。



10.1.2 从泛型类派生子类

当创建了带泛型声明的接口、父类之后，可以为该接口创建实现类，或者从该父类派生子类。需要指出的是，当使用这些接口、父类时不能再包含泛型形参。例如，下面代码就是错误的（Apple 类必须有 open 修饰才能派生子类）。

```
// 定义类 A 继承 Apple 类，Apple 类不能还使用泛型形参
class A: Apple<T>()
```

方法中的形参代表变量、常量、表达式等数据，本书把它们直接称为形参，或者称为数据形参。在定义方法时可以声明数据形参，在调用方法（使用方法）时必须为这些数据形参传入实际的数据。与此类似的是，在定义类、接口、方法时可以声明泛型形参，在使用类、接口、方法时应该为泛型形参传入实际的类型。

如果想从 Apple 类派生一个子类，则可以改为如下代码：

```
class A: Apple<String>()
```

与 Java 不同的是，Kotlin 要求始终为泛型参数明确地指定类型，而不管是通过显式指定，还是让系统进行推断。例如，如下代码是错误的。

```
// 系统无法推断出 T 是何种类型，因此编译报错
var a4 = Apple()
```

下面派生的子类也是错误的，因此系统无法推断出 Apple 的泛型参数 T 的类型。

```
// 使用 Apple 类时，没有为泛型 T 传入实际的类型参数，编译报错
public class A extends Apple
```

如果从 Apple<String>类派生子类，则在 Apple 类中所有使用泛型 T 的地方都将被替换成 String 类型，即它的子类将会继承 String 类型的 info 属性。如果子类需要重写父类的属性或方法，就必须注意这一点。下面程序示范了这一点。

程序清单：codes\10\10.2\A1.kt

```
class A1: Apple<String>(){
    // 正确重写了父类的属性
    // 属性类型与父类 Apple<String>的属性类型完全相同
    override var info: String? = null
        get() = "子类" + super.info
    /*
    // 下面方法是错误的，重写父类方法时返回值类型不一致
    override var info: Object? = null
    */
}
```

10.2 型变

Java 的泛型是不支持型变的，Java 采用通配符来解决这个问题；而 Kotlin 则采用安全的型变代替了 Java 的通配符。

10.2.1 泛型型变的需要

首先回顾一下 Java 泛型的特征：Java 的泛型是不支持型变的。通俗地说，List<String>并不是 List<Object>的子类，因此 List<String>不能直接赋值给 List<Object>。

Java 泛型不支持型变是有道理的，假如 Java 泛型支持型变，就会带来如下代码。

程序清单：codes\10\10.2\JavaNoVariance.java

```
import java.util.*;
public class JavaNoVariance
{
    public static void main(String[] args)
    {
        List<String> strs = new ArrayList<>();
        // 下面代码实际上会导致编译错误
        List<Object> objs = strs;
        // 如果上面代码可以通过，这行代码就会引发异常
        // 因为 objs 引用的集合元素实际上必须是 String
        objs.add(20);  // ①
    }
}
```

Java 泛型不支持型变主要体现在上面程序中的粗体字代码上：List<String>对象不能直接赋值给 List<Object>变量，否则①号代码就会引发错误。而 Java 泛型的设计原则是，如果一个程序可以无警告地通过编译，那么该程序运行时就不应该发生 ClassCastException 异常，因此 Java 这样设计确实可以保证程序的安全。

但 Java 取消型变之后，程序变得非常麻烦。比如 Java 的 Collection 中有一个 addAll()方法，该方法负责将另一个集合中的所有元素添加到本集合内。假如 Java 将该方法定义为如下形式：

```
interface Collection<E> {
    void addAll(Collection<E> items);
}
```

那么如下代码也是不能运行的。

```
Set<Number> numSet = new HashSet<>();
Set<Integer> intSet = new HashSet<>();
numSet.addAll(intSet);
```

如果不支持型变，上面粗体字代码的方法调用者是 Set<Number>，它要求其参数也必须是 Collection<Number>，而 Set<Integer>将不能作为参数传入。

但实际上我们知道上面代码绝对是安全的，对于 Set<Number>集合，程序完全可以将 Set<Integer>的集合内的元素安全地添加进去——这就是型变的需要。

为了处理型变的需要，Java 采用通配符方式进行处理，它将 addAll()方法定义为如下形式：

```
addAll(Collection<? extends E> c)
```

此时 addAll()方法的参数类型是指定上限的类型，其本质就是为了支持型变，因此如下代码可以正常运行。

程序清单：codes\10\10.2\WildVarianceExtends.java

```
import java.util.*;
public class WildCardVarianceExtends {
    public static void main(String[] args) {
        Set<Number> numSet = new HashSet<>();
        Set<Integer> intSet = new HashSet<>();
        intSet.add(2);
        intSet.add(3);
        // addAll(Collection<? extends E> c)使用?支持型变
        numSet.addAll(intSet); // 此处支持型变：intSet能被当成numSet
        Set<Double> doubleSet = new HashSet<>();
        doubleSet.add(2.3);
        doubleSet.add(4.3);
        // addAll(Collection<? extends E> c)使用?支持型变
        numSet.addAll(doubleSet); // 此处支持型变：doubleSet能被当成numSet
        System.out.println(numSet);
    }
}
```

上面两行粗体字代码是完全正常的，此时 addAll()方法调用者是 Set<Number>，因此该方法的参数可以是 Collection<Integer>、Collection<Double>等，只要其泛型参数是 Number 的子类即可。这种型变被称为“covariant”（中文翻译为“协变”）——Java 将这种语法称为“通配符上限”。

上面 addAll()方法的功能是从被作为参数的集合（支持协变的集合）中取出元素，再将集合元素添加到当前集合内。

对于“通配符上限”语法而言，从该集合对象中“拿出”对象总是安全的。比如 Collection<? extends Number>集合，从其中取出的元素至少是 Number 的子类。但程序不能向集合中添加元素（不能传入对象）。比如 Collection<? extends Number>集合，我们无法预测该集合实际引用的是 Collection<Integer>还是 Collection<Double>。



提示：

Java 泛型的通配符语法默认是有上限的，只不过其上限是 Object 类。

假如需要把对象传入支持型变的泛型类，此时就需要使用通配符下限。例如如下代码。

程序清单：codes\10\10.2\WildVarianceSuper.java

```
import java.util.*;
import java.util.function.*;
public class WildCardVarianceSuper {
    public static void main(String[] args) {
        // 定义一个Predicate对象
        Predicate<Number> pred = it -> it.doubleValue() > 10;
        Set<Integer> intSet = new HashSet<>();
        intSet.add(2);
        intSet.add(9);
        intSet.add(32);
        intSet.add(15);
        // removeIf(Predicate<? super E> filter)使用?支持型变
        intSet.removeIf(pred); // 使用Predicate<Number>对Set<Int>进行过滤
        System.out.println(intSet);
        Set<Double> doubleSet = new HashSet<>();
```

```
        doubleSet.add(2.5);
        doubleSet.add(14.4);
        doubleSet.add(3.8);
        doubleSet.add(15.8);
        // removeIf(Predicate<? super E> filter)使用?支持型变
        doubleSet.removeIf(pred); // 使用 Predicate<Number>对 Set<Double>进行过滤
        System.out.println(doubleSet);
    }
}
```

上面两行粗体字代码也是完全正常的，此时 removeIf()方法调用者是 Set<Integer>或 Set<Double>，该方法的参数完全可以是 Predicate<Number>、Predicate<Object>等，只要泛型参数是 Integer 或 Double 的父类即可。这种型变被称为“contravariance”（中文翻译为“逆变”）——Java 将这种语法称为“通配符下限”。

可见，Java 的“通配符下限”的本质也是为了支持型变，这种型变被细化为逆变。

上面 removeIf()方法的功能是将当前集合中的元素传入 Predicate 对象进行测试，符合该 Predicate 条件的元素将会被删除。

对于“通配符下限”语法而言，将对象传给泛型对象是安全的。比如 Predicate<? super Integer>对象，由于该 Predicate<T>中的泛型参数一定是 Integer 的父类，因此程序总可以向 Predicate 对象传入 Integer 值。但程序从泛型中取出对象是不安全的。比如 Predicate<? super Integer>，我们无法预测该集合实际引用的是 Predicate<Number>还是 Predicate<Object>。

简而言之，泛型存在如下规律。

- 通配符上限（泛型协变）意味着从中取出（out）对象是安全的，但传入对象（in）则不可靠。
- 通配符下限（泛型逆变）意味着向其中传入（in）对象是安全的，但取出对象（out）则不可靠。

Kotlin 利用上面两个规律，抛弃了泛型通配符语法，而是利用 in、out 来让泛型支持型变。

10.2.2 声明处型变

Kotlin 处理泛型型变的规则很简单。

- 如果泛型只需要出现在方法的返回值声明中（不出现在形参声明中），那么该方法就只是取出泛型对象，因此该方法就支持泛型协变（相当于通配符上限）；如果一个类的所有方法都支持泛型协变，那么该类的泛型参数可使用 out 修饰。
- 如果泛型只需要出现在方法的形参声明中（不出现在返回值声明中），那么该方法就只是传入泛型对象，因此该方法就支持泛型逆变（相当于通配符下限）；如果一个类的所有方法都支持泛型逆变，那么该类的泛型参数可使用 in 修饰。

下面程序先定义一个支持泛型协变的类。

程序清单：codes\10\10.2\User.kt

```
class User<out T>{
    // 此处不能用 var，否则就有 setter 方法
    // setter 方法会导致 T 出现在方法形参中
    val info: T
    constructor(info: T){
        this.info = info
    }
    fun test(): T{
```

```
        println("执行 test 方法")
        return info
    }
}
fun main(args: Array<String>) {
    // 此时 T 的类型是 String
    var user = User<String>("fkit")
    println(user.info)
    // 对于 u2 而言，它的类型是 User<Any>，此时 T 的类型是 Any
    // 由于程序声明了 T 支持协变，因此 User<String>可当成 User<Any>使用
    var u2: User<Any> = user // ①
}
```

上面程序中粗体字代码声明了一个泛型类，且使用了 out 修饰泛型形参，因此在该 User 类的内部，T 只能出现在方法的返回值声明中，不能出现在方法的形参声明中。所以，如果用 T 为 User 类声明属性，则只能声明为只读属性——否则，setter 方法的形参类型是 T，这就不符合要求了。

一旦声明了泛型类支持协变，程序即可安全地将 User<String>、User<Int>赋值给 User<Any>，只要尖括号中的类型是 Any 的子类即可，如上面程序中①号代码所示。

下面程序再定义一个支持泛型逆变的类。

程序清单：codes\10\10.2\Item.kt

```
class Item<in T>{
    fun foo(t: T){
        println(t)
    }
}
fun main(args: Array<String>) {
    // 此时 T 的类型是 Any
    var item = Item<Any>()
    item.foo(20)
    var im2: Item<String> = item
    // im2 的实际类型是 Item<Any>，因此它的 foo 参数只要是 Any 即可
    // 但我们声明了 im2 的类型为 Item<String>
    // 因此传入的参数只可能是 String，所以程序肯定是安全的
    im2.foo("Kotlin") // ①
}
```

上面程序中粗体字代码声明了一个泛型类，且使用了 in 修饰泛型形参，因此在该 Item 类的内部，T 只能出现在方法的形参声明中，不能出现在方法的返回值声明中。

一旦声明了泛型类支持逆变，程序即可安全地将 Item<Any>、Item<CharSequence>赋值给 User<String>，只要尖括号中的类型是 String 的父类即可，如上面程序中①号代码所示。

通过上面介绍不难发现，Kotlin 的处理规则很简单。

- 如果泛型 T（或其他字母）只出现在该类的方法的返回值声明中（T 代表的是传出值），那么该泛型形参即可使用 out 修饰 T。
- 如果泛型 T（或其他字母）只出现在该类的方法的形参声明中（T 代表的是传入参数），那么该泛型形参即可使用 in 修饰 T。

使用 out 修饰泛型的类支持协变，也就是可以将 User<String>、User<Int>当成 User<Any>处理，只要尖括号中的类型是 Any 的子类即可；使用 in 修饰泛型的类支持逆变，也就是可以将 Item<Any>、Item<CharSequence>当成 Item<String>处理，只要尖括号中的类型是 String 的

父类即可。

上面定义的 User、Item 类都是在声明时使用 out 或 in 指定泛型支持型变的，因此这种方式被称为“声明处型变”。

10.2.3 使用处型变：类型投影

声明时型变虽然方便，但它有一个限制：要么该类的所有方法都只用泛型声明返回值类型（此时可用 out 声明型变）；要么所有方法都只用泛型声明形参类型（此时可用 in 声明型变）。

如果一个类中有的方法使用泛型声明返回值类型，有的方法使用泛型声明形参类型，那么该类就不能使用声明处型变。典型的例子就是 Kotlin 的 Array 类，它无法使用声明处型变。因此该 Array 类包含如下两个方法：

```
class Array<T>(val size: Int) {
    fun get(index: Int): T { ///* ..... */ }
    fun set(index: Int, value: T) { ///* ..... */ }
}
```

上面 Array 类的泛型参数 T 既要出现在 get()方法的返回值声明中，也要出现在 set()方法的形参声明中，因此该 Array 类的泛型 T 既不能用 out 修饰，也不能用 in 修饰。

而 List 集合则不同，由于 List 集合是一个只读集合，程序只需要从 List 集合中取出元素（不能添加元素），因此 T 只会出现在 List 集合方法的返回值声明中。所以，List 集合可定义为支持协变。List 的源代码片段如下：

```
public interface List<out E> : Collection<E> {
    ...
}
```

如果不能使用声明处型变，则还可使用 Kotlin 提供的“使用处型变”。所谓使用处型变，就是在使用泛型时对其使用 out 或 in 修饰。

由于 Array 类本身不支持声明处型变，因此这里将会以 Array 为例来讲解使用处型变。下面先看使用处协变（使用 out 修饰）。

程序清单：codes\10\10.2\OutTest.kt

```
fun copy(from: Array<out Any>, to: Array<Any>) {
    val len = if(from.size < to.size) from.size else to.size
    for (i in 0 until len) {
        to[i] = from[i]
    }
}
fun main(args: Array<String>) {
    var arr1 = arrayOf(2, 3, 15, 26)
    var arr2: Array<Any> = arrayOf(4, 13, 25, 74, 34)
    copy(arr1, arr2)  // ①
    println(arr2.contentToString())
}
```

留意上面程序中 copy()函数的 from 参数的声明，该 from 参数的类型是 Array<out Any>，这就是使用处协变。也就是说，程序传入该 from 参数的可以是 Array<Int>、Array<String>等各种类型，只要尖括号中的类型是 Any 的子类即可，因此程序在①号代码处指定传给 from 参数的是 Array<Int>。

需要说明的是，如果将 from 参数声明为 Array<out Any>类型，那么就意味着只能安全地

从该 from 参数代表的数组中取元素，而不能将元素添加到 from 数组中——道理很明显：我们无法预测实际传给 from 参数的是 Array<Int>还是 Array<String>。

由于 Array 没有提供声明处型变，因此如下代码也是错误的。

```
var numArr: Array<Number> = arrayOf(2, 5, 1.2, 4.3)
println(numArr.contentToString())
numArr.set(0, 3.4)  // ②
var intArr: Array<Int> = arrayOf(3, 4, 5)
numArr = intArr; // ③ Array 不支持声明处型变，编译错误
println(numArr.contentToString())
```

上面程序中粗体字代码在使用 Array<Number>时没有声明型变，因此程序在③号代码处报错：Array<Int>不能当成 Array<Number>使用。

为了让③号代码能正常通过编译，我们可将粗体字代码改为如下形式：

```
var numArr: Array<out Number> = arrayOf(2, 5, 1.2, 4.3)
```

上面粗体字代码使用 out 修饰了 Number，这意味着它可以接受协变，因此程序可以将 Array<Int>、Array<String>等各种类型赋值给 numArr 变量，这样③号代码即可正常编译、运行。

由于 Array<out Number>只能保证从中取出的元素是 Number，程序不能向其中添加元素，因此上面程序中②号代码就会出错。



提示： 简单来说，Array<out Number>相当于 Java 的泛型上限：Array<? extends Number>，因此这种泛型型变用于支持协变，但不能向其中添加元素。

下面我们以 Array 为例来讲解使用处逆变（使用 in 修饰）。

程序清单：codes\10\10.2\InTest.kt

```
fun fill(dest: Array<in String>, value: String) {
    if (dest.size > 0) {
        dest[0] = value
    }
}
fun main(args: Array<String>) {
    var arr1: Array<CharSequence> = arrayOf("a", "b", StringBuilder("cc"))
    fill(arr1, "xxxx") // ①
    println(arr1.contentToString())

    var intArr: Array<in Int> = arrayOf(2, 5, 39)
    println(intArr.contentToString())
    intArr.set(0, 34)  // ②
    var numArr: Array<Number> = arrayOf(3, 4.1, 10.4)
    intArr = numArr; // ③ Array 不支持声明处型变，编译错误
    println(intArr.contentToString())
}
```

留意上面程序中 fill()函数的 dest 参数的声明，该 dest 参数的类型是 Array<in String>，这就是使用处逆变。也就是说，程序传入该 from 参数的可以是 Array<CharSequence>、Array<Any>等各种类型，只要尖括号中的类型是 String 的父类即可，因此程序在①号代码处指定传给 dest 参数的是 Array<CharSequence >。

需要说明的是，如果将 dest 参数声明为 Array<in String>类型，那么就意味着只能安全地向该 dest 参数代表的数组中添加元素，而不能从 dest 数组中取元素——道理很明显：我们无法预测实际传给 dest 参数的是 Array<CharSequence>还是 Array<Any>。

由于 Array 没有提供声明处型变，因此如下代码也是错误的。

```
var intArr: Array<Int> = arrayOf(2, 5, 39)
println(intArr.contentToString())
intArr.set(0, 34)  // ②
var numArr: Array<Number> = arrayOf(3, 4.1, 10.4)
intArr = numArr; // ③ Array 不支持声明处型变，编译错误
println(intArr.contentToString())
```

上面程序中粗体字代码在使用 Array<Int>时没有声明型变，因此程序在③号代码处报错：Array<Number>不能当成 Array<Int>使用。

为了让③号代码能正常通过编译，我们可将粗体字代码改为如下形式：

```
var intArr: Array<in Int> = arrayOf(2, 5, 39)
```

上面粗体字代码使用 in 修饰了 Int，这意味着它可以接受逆变，因此程序可将 Array<Number>、Array<Any>等各种类型赋值给 intArr 变量，这样③号代码即可正常编译、运行。

由于 Array<in Int>还可保证安全地向其中添加元素（由于它引用的 Array 集合元素一定是 Int 或 Int 的父类），因此上面程序中②号代码也可正常编译、运行。



提示：

简单来说，Array<in Int>相当于 Java 的泛型下限：Array<? super Int>，因此这种泛型型变用于支持逆变，还可保证向其中安全地添加元素。

10.2.4 星号投影

星号投影是为了处理 Java 的原始类型，比如如下 Java 代码：

```
ArrayList list = new ArrayList();
```

虽然 Java 的 List、ArrayList 都有泛型声明，但程序并没有为它们传入类型参数，这在 Java 程序中是允许的。这种用法被称为“原始类型”。

但在 Kotlin 中要写成如下形式。

程序清单：codes\10\10.2\StarProjection.kt

```
fun main(args: Array<String>) {
    //<*>必不可少，相当于 Java 的原始类型
    var list: ArrayList<*> = arrayListOf(1, "Kotlin")
    println(list)
}
```

上面的写法就是星号投影。

关于星号投影，下面给出一些示例说明。

- 假如定义了支持声明时型变的 Foo<out T>类，该泛型支持声明时协变，因此其中 T 是一个具有上限的协变类型参数，Foo<*>等价于 Foo<out Any?>。这意味着当 T 未知时，我们可以安全地从 Foo<*>读取 Any?类型的值。

- 假如定义了支持声明时型变的 Foo<in T>类，该泛型支持声明时逆变，因此其中 T 是一个逆变类型参数，Foo<*>等价于 Foo<in Nothing>。这意味着当 T 未知时，我们不能以任何安全的方式向 Foo <*>写入值。
- 假如定义了不支持声明时型变的 Foo<T>类，该泛型不支持型变。这意味着当 T 未知时，Foo<*>在读取值时等价于 Foo<out Any?>，在写入值时等价于 Foo<in Nothing>（即不能以任何安全的方式向 Foo <*>写入值）。

如果泛型类型具有多个类型参数，那么每个类型参数都可以单独指星号投影。假如声明了支持两个泛型参数的 Foo<in T, out U>类型，则关于星号投影的解释以下。

- 对于 Foo<*, String>，其实相当于 Foo<in Nothing, String>。
- 对于 Foo<Int, *>，其实相当于 Foo<Int, out Any?>。
- 对于 Foo<*, *>，其实相当于 Foo<in Nothing, out Any?>。

10.3 泛型函数

前面介绍了在定义类、接口时可以使用泛型形参，在该类、接口的方法定义和属性定义中，这些泛型形参可被当成普通类型来用。在另外一些情况下，在定义类、接口时没有使用泛型形参，但在定义方法时想自己定义泛型形参，这也是可以的，Kotlin 提供了对泛型函数的支持。

10.3.1 泛型函数的使用

所谓泛型函数，就是在声明函数时允许定义一个或多个泛型形参，泛型形参要用尖括号括起来，整体放在 fun 与函数名之间。泛型函数的语法格式如下：

```
fun <T , S> 函数名(形参列表): 返回值类型{
    // 函数体...
}
```

把上面泛型函数的语法格式和普通函数的语法格式进行对比，不难发现泛型函数的函数签名比普通函数的函数签名多了泛型声明，函数形参声明以尖括号括起来，多个函数形参之间以逗号（,）隔开，所有的函数形参声明都放在 fun 关键字和函数名之间。

例如，如下程序示范了泛型函数的用法。

程序清单：codes\10\10.3\GenericFunction.kt

```
fun <T> copy(from: List<T>, to: MutableList<in T>){
    for(ele in from){
        to.add(ele)
    }
}
fun main(args: Array<String>) {
    var strList = listOf("Java", "Kotlin")
    var objList: MutableList<Any> = mutableListOf(2, 1.2, "Android")
    // 指定泛型函数的 T 为 String 类型
    copy<String>(strList , objList)  // ①
    println(objList)
    var intList = listOf(7, 13, 17, 19)
    // 不显式指定泛型函数的 T 的类型，系统推断出 T 为 Int 类型
    copy(intList , objList)  // ②
    println(objList)
}
```

上面程序中的粗体字代码在 fun 和 copy 函数名之间声明了泛型：<T>，这样即可在该函数的形参声明或返回值声明中使用 T 来代表类型。

声明了泛型函数之后，调用泛型函数时可以在函数名后用尖括号传入实际的类型，如上面程序中①号代码所示；也可以在调用泛型函数时不为泛型参数指定实际的类型，而是让系统自动推断出泛型参数的类型，如上面程序中②号代码所示。

泛型函数也可用于扩展函数，例如如下代码。

程序清单：codes\10\10.3\GenericExtendFunction.kt

```
// 为泛型形参 T 扩展方法
fun <T> T.toBookString(): String{
    return "《${this.toString()}》"
}
fun main(args: Array<String>) {
    val a = 2
    // 显式指定泛型函数的 T 为 Int 类型
    println(a.toBookString<Int>())
    // 不显式指定泛型函数的 T 的类型，系统推断出 T 为 Double 类型
    println(3.4.toBookString())
    val str = "疯狂 Kotlin 讲义"
    // 不显式指定泛型函数的 T 的类型，系统推断出 T 为 String 类型
    println(str.toBookString())
}
```

上面程序中粗体字代码为泛型形参 T 扩展了一个 toBookString()方法，由于这个 T 只是一个泛型形参，它可以代表任意类型，因此程序相当于为所有类型都扩展了一个 toBookString()方法。所以，程序在 main()函数中可以为 Int、Double、String 等各种对象调用 toBookString()方法。

10.3.2 具体化类型参数

Kotlin 允许在内联函数（使用 inline 修饰的函数）中使用 reified 修饰泛型形参，这样即可将该泛型形参变成一个具体化的类型参数。一旦将泛型形参变成具体化的类型参数，接下来在该函数中就可以像使用普通类型一样使用该类型参数，包括使用 is、as 等运算符。

例如，我们要从某个 List 集合中查找第一个指定类型的元素，由于程序需要根据指定类型来查找数据，所以最容易想到的做法是，定义一个类型来作为参数。例如定义如下函数。

程序清单：codes\10\10.3\ReifiedTest.kt

```
val db = listOf("Java", java.util.Date(),
    103, 2.3, '我')
fun <T> findData(clazz: Class<T>): T? {
    for (ele in db) {
        if (clazz.isInstance(ele)) {
            @Suppress("UNCHECKED_CAST")
            return ele as? T
        }
    }
    return null
}
```

上面程序中粗体字代码定义了一个 clazz: Class<T>形参，这意味着调用该函数时必须显式传入一个 Class 对象——也就是说，必须通过反射来传入参数。因此，程序必须通过如下两行

代码来调用 findData()函数。

```
println(findData(Integer::class.java))
println(findData(java.lang.Double::class.java))
```

上面两行代码确实可以实现本例的需要，但是这种方式未免太不优雅了，因为我们知道泛型形参本身就是类型参数，当程序调用该函数时完全可通过泛型形参来传入类型参数，何必还要通过函数的参数来传入类型呢？

此时就可考虑使用 reified 修饰内联函数的泛型形参，这样就可直接在函数中使用该类型形参，从而避免用户通过函数的参数来传入类型。例如，将上面的 findData()函数改为如下形式（程序清单同上）：

```
// 使用 reified 修饰泛型形参，使之成为具体化的类型参数
inline fun <reified T> findData(): T? {
    for (ele in db) {
        // 在函数中直接使用 T 作为普通类型
        if (ele is T) {  // ①
            return ele
        }
    }
    return null
}
```

从上面程序中可以看出，程序此时使用 reified 修饰了泛型形参 T，接下来程序即可在函数体内直接使用该泛型形参 T，如上面程序中①号代码所示。

定义了该 findData()函数之后，接下来程序调用该函数时只要通过尖括号传入泛型参数即可，无须为函数传入参数值。程序可通过如下代码来调用该函数：

```
println(findData<Int>())
println(findData<Double>())
```

对于使用 reified 修饰的具体化的类型参数，程序甚至可以在函数体内对该参数使用反射，就像它是一个真正的类型一样。例如如下函数：

```
inline fun <reified T> membersOf() = T::class.members
```

上面程序中粗体字代码直接对泛型形参 T 使用反射，通过 class 属性来获取该类对应的 Class 对象，这也是具体化类型参数的优势。

10.4 设定类型形参的上限

Kotlin 泛型不仅允许在使用通配符形参时设定上限，而且可以在定义类型形参时设定上限，用于表示传给该类型形参的实际类型要么是该上限类型，要么是该上限类型的子类。下面程序示范了这种用法。

程序清单：codes\10\10.4\Apple.kt

```
class Apple<T: Number>{
    var col: T
    constructor(col: T){
        this.col = col
    }
}
fun main(args: Array<String>) {
```

```
    // 显式指定泛型函数的 T 是 Int 类型
    var ai = Apple<Int>(2)
    // 显式指定泛型函数的 T 是 Double 类型
    var ad: Apple<Double> = Apple(3.3)
    // 下面代码将引发编译异常。下面代码试图把 String 类型传给 T 形参
    // 但 String 不是 Number 的子类型，所以引起编译错误
//    var ap: Apple<String> = Apple("Kotlin")    // ①
}
```

上面程序定义了一个 Apple 泛型类，该 Apple 类的类型形参的上限是 Number 类，这表明使用 Apple 类时为 T 形参传入的实际类型参数只能是 Number 或 Number 类的子类。上面程序在①处将引起编译错误，因为类型形参 T 的上限是 Number 类型，而此处传入的实际类型是 String 类型。它既不是 Number 类型，也不是 Number 类型的子类型，所以将会导致编译错误。

泛型函数的泛型也可指定上限。例如如下代码（程序清单同上）：

```
fun <T: Number> sum(vararg params: T): Double{
    var sum = 0.0
    for (p in params) {
        sum += p.toDouble()
    }
    return sum
}
```

上面泛型函数指定了泛型形参 T 必须是 Number 或其子类，因此传给该函数的泛型参数可以是 Int、Double 等类型。例如如下代码（程序清单同上）：

```
// 显式指定泛型函数的 T 是 Int 类型，Int 是 Number 的子类型
println(sum<Int>(2, 4, 5))
// 系统推断出 T 为 Double 类型， Double 是 Number 的子类型
println(sum(1.2, 4.4, 5.6))
```



提示：

前面在定义泛型时没有指定上限，其实它有一个默认的上限：Any?。

在一种更极端的情况下，程序需要为类型形参设定多个上限（至多有一个父类上限，可以有多个接口上限），表明该类型形参必须是其父类的子类（是父类本身也行），并且实现多个上限接口。

由于在尖括号中只能指定一个上限，所以，如果要为泛型指定多个上限，则需要使用单独的 where 子句。例如如下代码：

```
// 为 T 指定多个上限
class Apple<T>  where T : Comparable<T>, T : Cloneable {
}
// 为 T 指定多个上限
fun <T> cloneWhenGreater(list: List<T>, threshold: T)
    where T : Comparable<T>, T : Cloneable {
}
```

10.5 本章小结

本章主要介绍了 Kotlin 提供的泛型支持。本章详细讲解了如何定义泛型接口、泛型类，

以及如何从泛型类、泛型接口派生子类或实现类，并深入讲解了泛型类的实质。本章详细介绍了 Kotlin 泛型的型变机制，包括声明处型变（包括协变和逆变两种）、使用处型变（也包括协变和逆变两种）、星号投影等机制。其中声明处型变是 Kotlin 独有的机制，Java 并不支持这种用法。使用处型变则和 Java 的通配符完全对应（或者说 Java 的通配符机制就是使用处型变）——Kotlin 的 Array<out Number>相当于 Java 的泛型上限：Array<? extends Number>，这属于使用处协变；Kotlin 的 Array<in Number>相当于 Java 的泛型下限：Array<? super Number>，这属于使用处逆变。本章还介绍了泛型函数的知识，泛型函数其实和 Java 的泛型方法大致相似。本章最后介绍了类型形参上限的用法，包括一个上限和多个上限两种用法。

CHAPTER

11

第 11 章 注解

本章要点

- 自定义注解
- 注解的属性和构造器
- @Retention 注解的功能和用法
- @Target 注解的功能和用法
- @MustBeDocumented 注解的功能和用法
- @Repeatable 注解的功能和用法
- 提取、使用注解信息
- 指定注解的作用目标
- 使用 Java 注解

Kotlin 的注解与 Javar 的注解完全相同，也是代码里的特殊标记，这些标记可以在编译、类加载、运行时被读取，并执行相应的处理。通过使用注解，程序开发人员可以在不改变原有逻辑的情况下，在源文件中嵌入一些补充信息。代码分析工具、开发工具和部署工具都可以通过这些补充信息进行验证或者进行部署。

通过在程序中添加注解来为程序提供一些额外的补充信息，这些补充信息通常是某种配置文件，以前往往使用 XML 来完成，现在很多地方会采用注解来代替 XML。而 Kotlin 也提供了一些 API 来读取、使用程序中的注解信息，这些内容都将在本章为读者一一介绍。

11.1 Kotlin 注解入门

Kotlin 使用 annotation class 关键字来定义注解。下面将详细介绍 Kotlin 注解的用法

11.1.1 定义注解

Kotlin 使用 annotation class 关键字（就像使用 enum class 定义枚举类一样），定义注解非常简单，Kotlin 甚至不允许为注解定义注解体，也就是说，注解后面不能有花括号。

例如，如下代码定义了一个简单的注解。

程序清单：codes\11\11.1\SimpleAnnotation.kt

```
// 定义一个简单的注解
annotation class Test
```

定义了该注解之后，就可以在程序的任何地方使用该注解。使用注解的语法非常类似于使用 public、final 这样的修饰符，通常可用于修饰程序中的类、方法、属性、接口等定义。通常会把注解放在所有修饰符之前，而且由于使用注解时可能还需要为属性指定值，因此注解的长度可能较长，所以有时候也会把注解另放一行，如下面程序所示。

```
// 使用@Test 修饰类定义
@Test class MyClass{
    ...
}
```

在默认情况下，注解可用于修饰任何程序元素，包括类、接口、属性、方法等。例如，如下程序使用@Test 来修饰类、属性、方法（程序清单同上）。

```
// 使用@Test 修饰类定义
@Test class MyClass{
    // 使用@Test 注解修饰属性
    @Test var name: String = ""
    // 使用@Test 注解修饰方法
    @Test fun info(){
        ...
    }
    ...
}
```

如果要用注解来修饰主构造器，就像前面所介绍的，程序必须为主构造器添加 constructor 关键字。例如如下代码（程序清单同上）：

```
class User @Test constructor(var name: String, var pass: String){
}
```

11.1.2 注解的属性和构造器

注解还可以带属性，由于注解没有注解体，因此注解的属性只能在注解声明部分指定。实际上，相当于在注解的主构造器中指定注解的属性。

由于注解与普通类不同，注解的属性值只能在使用时指定；并且一旦为注解的属性指定了属性值，以后就绝对不会改变其属性值，因此注解的属性只能定义为只读属性。

例如，如下代码定义了一个带属性和构造器的注解。

程序清单：codes\11\11.1\PropertyAnnotation.kt

```
annotation class MyTag(val name: String,
    val age: Int)
```

使用 annotation class 定义的注解其实就相当于定义了一个注解接口，这个注解接口继承了 kotlin.Annotation 接口。

需要说明的是，注解的属性不能使用可空类型（不能在类型后添加“?”），这是因为 JVM 本身不允许使用 null 作为注解的属性值。

注解的属性可支持如下类型。

- 对应于 Java 基本类型的类（如 Int、Long 等）
- 字符串
- 类（如 Foo::class）
- 枚举
- 其他注解
- 上面各种类型的数组

一旦在注解中定义了属性之后，使用该属性时就应该为其指定属性值，如下面代码所示（程序清单同上）。

```
class Item {
    // 使用带属性的注解时，需要为属性指定属性值
    @MyTag(name="xx", age=6)
    fun info(){
        ...
    }
    ...
}
```

也可以在定义注解的属性时使用等号（=）为其指定初始值（默认值）（就像定义类时在主构造器中为类的属性指定初始值一样），注解的初始值只能是编译时常量。

例如，如下代码定义了@MyTag 注解，该注解中包含了两个属性：name 和 age，这两个属性使用等号（=）指定了初始值。

程序清单：codes\11\11.1\DefaultValueAnnotation.kt

```
annotation class MyTag(val name: String = "yeeku",
    val age: Int = 29)
```

如果为注解的属性指定了默认值，那么在使用该注解时可以不为这些属性指定值，而是直接使用默认值。例如如下程序（程序清单同上）：

```
class Item {
    // 使用带属性的注解
    // 因为它的属性有默认值，所以可以不为该属性指定值
```

```
    @MyTag
    fun info(){
        ...
    }
    ...
}
```

当然，也可以在使用@MyTag 注解时为属性指定值。如果为@MyTag 的属性指定了值，则其默认值不会起作用。

根据注解是否可以包含属性，可以把注解分为如下两类。

- **标记注解**：没有定义属性的注解被称为标记注解。这种注解仅利用自身的存在与否来提供信息，如前面所介绍的@Test 等注解。
- **元数据注解**：包含属性的注解被称为元数据注解。因此它们可以接受更多的配置信息（以属性值的方式进行设置）。如前面所介绍的@MyTag 等注解。

与 Java 类似的是，如果注解的属性名为 value，则为 value 属性指定属性值时可省略属性名。例如如下程序。

程序清单：codes\11\11.1\ValuePropertyAnnotation.kt

```
// 定义带 value 属性的注解
annotation class MyTag(val value:String)

class Category {
    // 为 value 属性指定属性值时无须带属性名
    @MyTag("yeeku")
    fun info(){
        ...
    }
    ...
}
```

Kotlin 使用 vararg 修饰需要指定多个值的属性（相当于数组类型的属性）。例如如下程序。

程序清单：codes\11\11.1\ArrayPropertyAnnotation.kt

```
annotation class MyTag(vararg val infos: String)

class Role{
    // 使用带属性的注解
    // 因为它的属性有默认值，所以可以不为该属性指定值
    @MyTag("java", "yeeku", "kotlin")  // ①
    fun info(){
        ...
    }
    ...
}
```

对于使用 vararg 修饰的属性，不管它的属性名是什么，系统为该属性指定属性值时，都不需要指定属性名，直接在注解中指定多个属性值即可。如上面程序中①号代码所示。

如果将一个注解作为另一个注解的属性值，那么在使用注解时不需要以@作为前缀。例如，如下程序示范了将一个注解作为另一个注解的属性值。

程序清单：codes\11\11.1\AnnotationTypeProperty.kt

```
// 定义带 value 属性的注解
annotation class MyTag(val value:String)

// 该注解的 target 属性的类型是 MyTag
annotation class ShowTag(val message: String,
    val target: MyTag)

@ShowTag("message 属性值", target = MyTag("yeeku"))
class Circle
```

如果需要将一个类作为注解的属性，请使用 Kotlin 类（KClass），Kotlin 编译器会自动将其转换为 Java 类，以便 Java 代码能够正常看到该注解和参数。

例如，如下代码示范了使用类作为注解的属性。

程序清单：codes\11\11.1\ClassTypeProperty.kt

```
import kotlin.reflect.*
// arg1 的类型是 KClass<*>，这是星号投影用法，相当于 Java 的原始类型
// arg2 的类型是 KClass<out Any>，这是使用处协变的用法
// 可传入 KClass<Int>、KClass<String>等，只要尖括号里的类型是 Any 的子类即可
annotation class DrawTag(val arg1: KClass<*>,
    val arg2: KClass<out Any>)
@DrawTag(arg1=String::class, arg2=Int::class)
class Circle
```

11.2 元注解

Kotlin 在 kotlin.annotation 包下提供了 4 个 Meta 注解（元注解），这 4 个元注解都用于修饰其他的注解定义。

11.2.1 使用@Retention

@Retention 只能修饰注解定义，用于指定被修饰的注解可以保留多长时间。@Retention 元注解包含一个 AnnotationRetention 类型的 value 属性，所以使用@Retention 时必须为该 value 属性指定值。

value 属性的值只能是如下 3 个。

- AnnotationRetention.SOURCE：注解只保留在源代码中，编译器直接丢弃这种注解。
- AnnotationRetention.BINARY：编译器将把注解记录在 class 文件中。当运行该字节码文件时，JVM 不可获取注解信息。
- AnnotationRetention.RUNTIME：编译器将把注解记录在 class 文件中。当运行该字节码文件时，JVM 也可获取注解信息，程序可以通过反射获取该注解信息。这是默认值。

如果要通过反射获取注解信息，就需要使用 value 属性值为 AnnotationRetention.RUNTIME 的@Retention（或省略该元注解）。使用@Retention 元注解可采用如下代码为 value 指定值。

```
//下面定义的 Testable 注解保留到运行时
@Retention(value=AnnotationRetention.RUNTIME)
annotation class Testable
```

也可采用如下代码来为 value 指定值。

```
//下面定义的 Drawable 注解将被编译器直接丢弃
@Retention(AnnotationRetention.SOURCE)
annotation class Drawable
```

上面代码中使用@Retention 元注解时，并未通过 value=AnnotationRetention.SOURCE 的方式来为该属性指定值，这是因为当注解的属性名为 value 时，在程序中可以直接在注解后的括号里指定该属性的值，而无须使用 name=value 的形式。

注意：

如果使用注解时只需要为 value 属性指定值，则可以直接在该注解后的括号里指定 value 属性的值，而无须使用“value=属性值”的形式。



11.2.2 使用@Target

@Target 也只能修饰注解定义，用于指定被修饰的注解能修饰哪些程序单元。@Target 元注解包含一个类型为 AnnotationTarget 数组的 allowedTargets 属性，该属性的值只能是如下几个值组成的数组。

- AnnotationTarget.CLASS：指定该策略的注解只能修饰类。
- AnnotationTarget.ANNOTATION_CLASS：指定该策略的注解只能修饰注解。
- AnnotationTarget.TYPE_PARAMETER：指定该策略的注解只能修饰泛型形参（目前暂时还不支持）。
- AnnotationTarget.PROPERTY：指定该策略的注解只能修饰属性。
- AnnotationTarget.FIELD：指定该策略的注解只能修饰字段（包括属性的幕后字段）。
- AnnotationTarget.LOCAL_VARIABLE：指定该策略的注解只能修饰局部变量。
- AnnotationTarget.VALUE_PARAMETER：指定该策略的注解只能修饰函数或构造器的形参。
- AnnotationTarget.CONSTRUCTOR：指定该策略的注解只能修饰构造器。
- AnnotationTarget.FUNCTION：指定该策略的注解只能修饰函数和方法（不包含构造器）。
- AnnotationTarget.PROPERTY_GETTER：指定该策略的注解只能修饰属性的 getter 方法。
- AnnotationTarget.PROPERTY_SETTER：指定该策略的注解只能修饰属性的 setter 方法。
- AnnotationTarget.TYPE：指定该策略的注解只能修饰类型。
- AnnotationTarget.EXPRESSION：指定该策略的注解只能修饰各种表达式。
- AnnotationTarget.FILE：指定该策略的注解只能修饰文件。
- AnnotationTarget.TYPEALIAS：指定该策略的注解只能修饰类型别名。

与使用@Retention 类似的是，使用@Target 也可以直接在括号里指定 value 值，而无须使用 name=value 的形式。如下代码指定@ActionListenerFor 注解只能修饰属性。

```
@Target(AnnotationTarget.PROPERTY)
annotation class ActionListenerFor
```

如下代码指定@Drawable 注解只能修饰函数和类。

```
@Target(allowedTargets={AnnotationTarget.FUNCTION,AnnotationTarget.CLASS})
```

```
annotation class Drawable
```

11.2.3 使用@MustBeDocumented

使用@MustBeDocumented 元注解修饰的注解将被文档工具提取到 API 文档中，如果定义注解类时使用了@MustBeDocumented 修饰，则所有使用该元注解修饰的程序元素的 API 文档中将会包含该注解说明。

下面代码定义了一个@Testable 注解，程序使用@MustBeDocumented 修饰该注解，所以@Testable 注解将被文档工具所提取。

程序清单：codes\11\11.2\Testable.kt

```
@Retention(AnnotationRetention.RUNTIME)
@Target(AnnotationTarget.FUNCTION)
// 定义@Testable 注解将被文档工具所提取
@MustBeDocumented
annotation class Testable
```

上面代码中的粗体字代码指定了文档工具生成 API 文档时将提取@Testable 的使用信息。下面代码定义了一个 MyTest 类，该类中的 info()方法使用了@Testable 修饰。

程序清单：codes\11\11.2\MyTest.kt

```
class MyTest{
    // 使用@Testable 修饰 info()方法
    @Testable
    fun info(){
        println("info 方法...")
    }
}
```

使用 Dokka 工具为 Testable.kt、MyTest.kt 文件生成的 API 文档如图 11.1 所示。

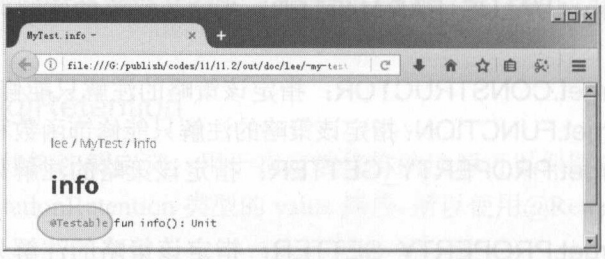


图 11.1 文档工具提取了有@MustBeDocumented 修饰的注解

如果把上面 Testable.kt 程序中的粗体字代码删除或注释掉，再次使用 Dokka 工具生成的 API 文档如图 11.2 所示。

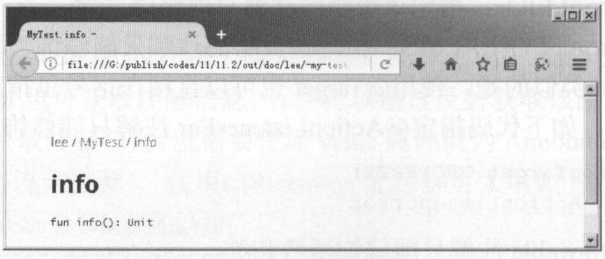


图 11.2 文档工具不提取没有@MustBeDocumented 修饰的注解

对比图 11.1 和 11.2 所示的两份 API 文档，发现图 11.1 中的 info()方法说明中包含了@Testable 的信息（见图中的灰色区域），这就是使用@MustBeDocumented 元注解的作用。

11.2.4 使用@Repeatable 标记可重复注解

Kotlin 允许使用多个相同的注解来修饰同一个程序单元，这种注解称为可重复注解。

开发可重复注解需要使用@Repeatable 修饰，下面通过示例来介绍如何开发可重复注解。首先定义一个@FkTag 注解。

程序清单：codes\11\11.2\FkTag.kt

```
@Retention(AnnotationRetention.SOURCE)
@Target(AnnotationTarget.CLASS)
// 指定该注解是可重复注解
@Repeatable
annotation class FkTag(val name: String = "疯狂软件",
    val age: Int)
```

上面定义了@FkTag 注解，该注解包含两个属性。程序还使用了@Repeatable 来修饰该注解，这意味着它是一个可重复注解，因此可直接使用多个@FkTag 注解修饰目标程序单元，如下面代码所示（程序清单同上）。

```
@FkTag(age=5)
@FkTag(name="疯狂 Java" , age=9)
class FkTagTest
```

需要说明的是，由于在 Java 8 之前 JVM 并不支持可重复注解，Kotlin 也没有办法突破该限制，因此可重复注解的@Retention 策略只能指定为 AnnotationRetention.SOURCE，这意味着可重复注解只能被 Kotlin 编译器读取，接下来 Kotlin 编译器会直接丢弃该注解信息。

注意：

在 Java 8 之前，可重复注解的@Retention 策略只能指定为 AnnotationRetention.SOURCE。



11.3 使用注解

Kotlin 的反射也提供了一些支持注解的 API，这些 API 用起来非常简单。

11.3.1 提取注解信息

使用注解修饰类、方法、属性等成员之后，这些注解不会自己生效，必须由开发者提供相应的工具来提取并处理注解信息。

Kotlin 使用 kotlin.Annotation 接口来代表程序元素前面的注解，该接口是所有注解的父接口。Kotlin 在 kotlin.reflect 包下新增了 KAnnotatedElement 接口，该接口代表程序中可以接受注解的程序元素。该接口主要有如下几个实现类。

- KCallable：代表可执行的程序实体，如函数和属性。
- KClass：代表 Kotlin 的类、接口等类型。
- KParameter：代表函数和属性的参数。

在 kotlin.reflect 包下主要包含一些实现反射功能的工具类，该包所提供的反射 API 包含了读取运行时注解的能力。只有当定义注解时使用了@Retention(AnnotationRetention.RUNTIME)修饰，该注解才会保留到程序运行时，JVM 才会在装载*.class 文件时读取保存在 class 文件中的注解。

KAnnotatedElement 接口是所有程序元素（如 KClass、KCallable、KParameter）的父接口，所以程序通过反射获取了某个程序单元对应的 KAnnotatedElement 对象（如 KClass、KCallable、KParameter）之后，程序就可以调用该对象的如下属性和方法来访问注解信息。

- annotations: List<Annotation>：该属性返回该程序单元上所有的注解。
- <T: Annotation> findAnnotation(): T?：根据注解类型返回该程序单元上特定类型的注解。如果该类型的注解不存在，则该方法返回 null。

注意：

为了获得程序中的程序元素（如 KClass、KFunction 等），必须使用反射知识。如果读者需要获得关于反射的更详细内容，则可以参考本书下一章的介绍。



下面代码片段用于获取 Test 类中修饰 info()方法的所有注解，并将这些注解打印出来。

```
val aArray = Test::info.annotations
// 遍历所有注解
for (an in aArray ){
    println(an)
}
```

如果需要获取某个注解里的元数据，则可以将注解转型成所需的注解类型，然后通过注解对象的属性来访问这些元数据。代码片段如下：

```
val anArr = Test::info.annotations
// 遍历每个注解对象
for (tag in anArr){
    // 如果 tag 注解是 MyTag1 类型
    if (tag is MyTag1){
        println("注解是：" + tag)
        // 输出注解的 prop1 和 prop2 两个属性的值
        println("tag.prop1：" + tag.prop1)
        println("tag.prop2：" + tag.prop2)
    }
    // 如果 tag 注解是 MyTag2 类型
    if (tag is MyTag2){
        println("注解是：" + tag)
        // 将 tag 强制类型转换为 MyTag2
        // 输出 tag 对象的 name 和 age 两个属性的值
        println("tag.name：" + tag.name)
        println("tag.age：" + tag.age)
    }
}
```

11.3.2 使用注解的示例

下面分别介绍两个使用注解的例子。第一个例子中的@Testable 注解没有任何属性，它仅是一个标记注解，其作用是标记哪些方法需要测试。

程序清单：codes\11\11.3\01\Testable.kt

```
/@Retention(AnnotationRetention.RUNTIME)
@Target(AnnotationTarget.FUNCTION)
// 定义一个标记注解，不包含任何属性
annotation class Testable
```

上面程序定义了一个@Testable 注解，在定义该注解时使用了@Retention 和@Target 两个元注解，其中@Retention 元注解指定@Testable 注解可以保留到运行时（JVM 可以提取到该注解的信息），而@Target 元注解指定@Testable 注解只能修饰函数（或方法）。

如下 MyTest 测试用例中定义了 8 个方法，这些方法没有太大的区别，其中 4 个方法使用@Testable 注解来标记它们是需要测试的。

程序清单：codes\11\11.3\01\MyTest.kt

```
class MyTest{
    // 使用@Testable 注解指定该方法是需要测试的
    @Testable
    fun m1(){
    }
    fun m2() {
    }
    // 使用@Testable 注解指定该方法是需要测试的
    @Testable
    fun m3() {
        throw new IllegalArgumentException("参数出错了！")
    }
    fun m4(){
    }
    // 使用@Testable 注解指定该方法是需要测试的
    @Testable
    fun m5(){
    }
    fun m6(){
    }
    // 使用@Testable 注解指定该方法是需要测试的
    @Testable
    fun m7(){
        throw new RuntimeException("程序业务出现异常！")
    }
    fun m8(){
    }
}
```

正如前面所提到的，仅仅使用注解来标记程序元素对程序是不会有任何影响的，这也是注解的一条重要原则。为了让程序中的这些注解起作用，接下来必须为这些注解提供一个注解处理工具。

下面的注解处理工具会分析目标类，如果目标类中的方法使用了@Testable 注解修饰，则通过反射来运行该测试方法。

程序清单：codes\11\11.3\01\ProcessorTest.kt

```
import kotlin.reflect.*
import kotlin.reflect.full.*
inline fun <reified T: Any> processTestable() {
    var passed = 0
```

```
    var failed = 0
    val target = T::class.createInstance<T>()
    // 遍历 T 对应的类里的所有方法
    for (m in T::class.functions) {
        // 如果该方法使用了@Testable 修饰
        if (m.findAnnotation<Testable>() != null){
            try {
                // 调用 m 方法
                m.call(target)
                // 测试成功，passed 计数器加 1
                passed++
            }catch (ex: Exception){
                println("方法" + m + "运行失败，异常："
                    + ex.cause)
                // 测试出现异常，failed 计数器加 1
                failed++
            }
        }
    }
    // 统计测试结果
    println("共运行了:" + (passed + failed)
        + "个方法，其中：\n" + "失败了:" + failed + "个，\n"
        + "成功了:" + passed + "个！")
}
fun main(args: Array<String>){
    // 处理 MyTest 类
    processTestable<MyTest>()
}
```

上面程序定义了一个<reified T: Any> processTestable()函数，该函数可接收一个泛型参数，分析该泛型参数所代表的类，并运行该目标类中使用@Testable 修饰的方法。



提示：

上面程序使用到 Kotlin 的反射知识，以及 Kotlin 1.1 对反射的进一步扩展，所以程序导入了 kotlin.reflect.*和 kotlin.reflect.full.*两个包。

该程序的 main()函数非常简单，使用 processTestable()函数分析目标类即可。

运行上面程序，会看到如下运行结果：

```
方法 fun MyTest.m3(): kotlin.Unit 运行失败，异常：java.lang.IllegalArgumentException: 参数出错了！
方法 fun MyTest.m7(): kotlin.Unit 运行失败，异常：java.lang.RuntimeException: 程序业务出现异常！
共运行了:4 个方法，其中：
失败了:2 个，
成功了:2 个！
```

通过这个运行结果可以看出，程序中的@Testable 起作用了，在 MyTest 类中使用@Testable 注解修饰的方法都被测试了。



提示：

通过上面例子读者不难看出，其实注解十分简单，它们就是对源代码增加的一些特殊标记，这些特殊标记可通过反射获取，当程序获取到这些特殊标记后可以做出相应的处理（当然也可以完全忽略这些注解）。

前面介绍的只是一个标记注解，程序通过判断该注解存在与否来决定是否运行指定方法。下面程序通过使用注解来简化事件编程。在传统的事件编程中总是需要通过 addActionListener() 方法来为事件源绑定事件监听器，本示例则通过@ActionListenerFor 来为程序中的按钮绑定事件监听器。

程序清单：codes\11\11.3\02\ActionListenerFor.kt

```
import kotlin.reflect.*
import java.awt.event.*
// 指定该注解只能修饰属性
@Target(AnnotationTarget.PROPERTY)
@Retention(AnnotationRetention.RUNTIME)
// 定义一个属性，用于设置元数据
// 该 listener 属性用于保存监听器实现类
annotation class ActionListenerFor(
    val listener: KClass<out ActionListener>)
```

定义了@ActionListenerFor 注解之后，使用该注解时需要为 listener 属性指定属性值，该属性用于指定监听器的实现类。下面程序使用@ActionListenerFor 注解来为两个按钮绑定事件监听器。

程序清单：codes\11\11.3\02\AnnotationTest.kt

```
import javax.swing.*
import java.awt.event.*
class AnnotationTest{
    val mainWin = JFrame("使用注解绑定事件监听器")
    // 使用注解为 ok 按钮绑定事件监听器
    @ActionListenerFor(listener=OkListener::class)
    val ok = JButton("确定")
    // 使用注解为 cancel 按钮绑定事件监听器
    @ActionListenerFor(listener=CancelListener::class)
    val cancel = JButton("取消")
    fun init(){
        // 初始化界面的方法
        val jp = JPanel()
        jp.add(ok)
        jp.add(cancel)
        mainWin.add(jp)
        processAnnotations(this)  // ①
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
        mainWin.pack()
        mainWin.setVisible(true)
    }
}
// 定义 ok 按钮的事件监听器实现类
class OkListener: ActionListener{
    override fun actionPerformed(evt: ActionEvent){
        JOptionPane.showMessageDialog(null, "单击了确认按钮")
    }
}
// 定义 cancel 按钮的事件监听器实现类
class CancelListener: ActionListener{
    override fun actionPerformed(evt: ActionEvent){
        JOptionPane.showMessageDialog(null, "单击了取消按钮")
    }
```

```
}
fun main(args: Array<String>) {
    AnnotationTest().init()
}
```

上面程序中的粗体字代码定义了两个 JButton 按钮，并使用@ActionListenerFor 注解为这两个按钮绑定了事件监听器。使用@ActionListenerFor 注解时传入了 listener 元数据，该元数据用于设定每个按钮的监听器实现类。

正如前面所提到的，如果仅在程序中使用注解是不会起任何作用的，必须使用注解处理工具来处理程序中的注解。程序中①号代码使用了 processAnnotations()函数来处理注解，该处理器分析目标对象中的所有属性，如果在属性前使用了@ActionListenerFor 修饰，则取出该注解中的 listener 元数据，并根据该元数据来绑定事件监听器。

程序清单：codes\11\11.3\02\ActionListenerInstaller.kt

```
import javax.swing.*
import kotlin.reflect.*
import kotlin.reflect.full.*
// 处理注解的方法，其中 obj 是包含注解的对象
fun processAnnotations(obj: Any){
    // 获取 obj 对象的类
    val cl = obj::class
    // 获取指定 obj 对象的所有成员，并遍历每个成员
    for (prop in cl.memberProperties){
        // 获取该成员上 ActionListenerFor 类型的注解
        val a = prop.findAnnotation<ActionListenerFor>()
        // 获取属性 prop 的值
        val fObj = prop.call(obj)
        // 如果 fObj 是 AbstractButton 的实例，且 a 不为 null
        if (a != null && fObj != null
            && fObj is AbstractButton){
            // 获取 a 注解的 listener 属性值（它是一个监听器类）
            val listenerClazz = a.listener
            // 使用反射来创建 listener 类的对象
            val al = listenerClazz.createInstance()
            // 为 fObj 按钮添加事件监听器
            fObj.addActionListener(al)
        }
    }
}
```

上面程序中的前两行粗体字代码根据@ActionListenerFor 注解的元数据取得了监听器实现类，然后通过反射来创建监听器对象；第三行粗体字代码将监听器对象绑定到指定的按钮（按钮由被@ActionListenerFor 修饰的属性代表）。

运行上面的 AnnotationTest 程序，将会看到如图 11.3 所示的窗口。

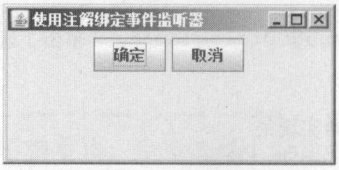


图 11.3 使用注解绑定事件监听器

单击“确定”按钮，将会弹出如图 11.4 所示的“单击了确认按钮”消息对话框，这表明

使用注解成功地为 ok、cancel 两个按钮绑定了事件监听器。

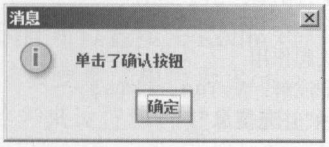


图 11.4 使用注解成功地绑定了事件监听器

11.4 Java 注解与 Kotlin 的兼容性

Java 注解与 Kotlin 完全兼容，只是在使用时略加注意即可。

11.4.1 指定注解的作用目标

根据前面的介绍我们知道，Kotlin 程序往往比较简洁，Kotlin 程序的一个程序单元有时候会变成 Java 的多个程序单元。比如：

- 带属性声明的主构造器会变成 Java 的成员变量定义、getter 方法、setter 方法（如果是读写属性）、构造器参数。
- 属性会变成 Java 的成员变量定义、getter 方法、setter 方法（如果是读写属性）。

这样就产生了一个问题：有时候我们只想用注解修饰特定的程序单元，比如只希望用注解修饰属性对应的幕后字段，或者只希望用注解修饰属性对应的 getter 方法，那该怎么办呢？

此时就需要为注解指定作用目标，语法格式如下：

```
@目标:注解(注解属性值)
```

如果在同一个目标上要指定多个注解，则需要将多个注解放在方括号中，并用空格隔开，语法格式如下：

```
@目标:[注解 1(注解属性值) 注解 2(注解属性值), ...]
```

从上面的语法格式不难看出，为注解指定作用目标，其实就是在@符号和注解之间添加目标名和冒号。Kotlin 支持的目标包含如下几个。

- file：指定注解对文件本身起作用。
- property：指定注解对整个属性起作用（这种目标的注解对 Java 不可见，因为 Java 并没有真正的属性）。
- field：指定注解对属性的幕后字段起作用。
- get：指定注解对属性的 getter 方法起作用。
- set：指定注解对属性的 setter 方法起作用。
- receiver：指定注解对扩展方法或扩展属性的接收者起作用。
- param：指定注解对构造器的参数起作用。
- setparam：指定注解对 setter 方法的参数起作用。
- delegate：指定注解对委托属性存储其委托实例的字段起作用。

下面先看一个简单的例子，程序指定注解只对属性的 getter 方法起作用。

程序清单：codes\11\11.4\TargetGetter.kt

```
import java.lang.reflect.*
// 先定义两个注解
```

```
annotation class MyTag
annotation class FkTag(val info: String)
class Item{
    // 指定注解只对 getter 方法起作用
    // 对 getter 方法应用了两个注解：MyTag、FkTag
    @get:[MyTag FkTag(info="补充信息")]
    var name: String = "fkjava"
}
fun main(args: Array<String>) {
    // 获取 Item 类对应的 Java 类（Class 对象）
    val clazz = Item::class.java
    // 遍历 clazz 类所包含的全部方法
    for (mtd in clazz.getDeclaredMethods()) {
        println("--方法${mtd}上的注解如下--")
        // 遍历该方法上直接声明的所有注解
        for(an in mtd.getDeclaredAnnotations()){
            println(an)
        }
    }
    // 遍历 clazz 类所包含的全部成员变量
    for (f in clazz.getDeclaredFields()) {
        println("--方法${f}上的注解如下--")
        // 遍历该成员变量上直接声明的所有注解
        for(an in f.getDeclaredAnnotations()){
            println(an)
        }
    }
}
```

上面程序中粗体字代码指定 name 属性的 getter 方法应用了两个注解，其实就是在原来的注解用法前增加@get:部分，与原来不指定目标的注解区别并不大。

程序后面的 main()函数主要就是分析 Item 类中各方法、成员变量的注解。由于要分析成员变量上的注解，因此 main()函数使用了 Java 的反射 API（由于 Kotlin 并不支持单独定义成员变量，因此 Kotlin 的反射 API 不支持直接操作成员变量），通过该 main()的运行可以看到程序中添加的两个注解只作用于属性的 getter 方法上。运行该程序，可看到如下结果：

```
--方法public final java.lang.String Item.getName()上的注解如下--
@MyTag()
@FkTag(info=补充信息)
--方法public final void Item.setName(java.lang.String)上的注解如下--
--方法private java.lang.String Item.name上的注解如下--
```

如果要指定注解作用于整个文件本身，则必须将注解放在 package 语句（如果有 package 语句）之前，或者所有导包语句之前（如果没有 package 语句）。代码如下：

```
// 指定@FileTag 注解作用于整个文件
@file:FileTag("yeeku")
package org.crazyit.demo
```

如果要指定注解作用于扩展方法或扩展属性的接收者，则使用带 receiver:的注解修饰整个扩展方法或扩展属性即可。例如如下代码：

```
// 指定@MyTag 注解作用于扩展方法的接收者（String）
fun @receiver:MyTag String.fun() { }
```

11.4.2 使用 Java 注解

Kotlin 完全兼容 Java 注解，因此可以直接在 Kotlin 程序中使用 Java 注解。

需要说明的是，Java 注解的成员变量（相当于 Kotlin 注解的属性，后文统称为“属性”）是没有顺序的，因此只能通过属性名来设置属性值（Kotlin 注解的属性还可通过位置来设置属性值）。

对比如下两个注解。下面先定义一个 Java 注解。

程序清单：codes\11\11.4\JavaTag.java

```
public @interface JavaTag{
    public String name();
    public int age();
}
```

下面再定义一个 Kotlin 注解。

程序清单：codes\11\11.4\KotlinTag.kt

```
annotation class KotlinTag(val name: String, val age: Int)
```

上面两个注解基本相同，它们都包含了 name 和 age 两个属性。

接下来在程序中使用这两个注解就可看到它们的区别。

程序清单：codes\11\11.4\JavaVsKotlin.java

```
// Kotlin 注解可通过位置来指定属性值
// 第一个值传给 name 属性，第二个值传给 age 属性
@KotlinTag("疯狂 Kotlin 讲义", 29)
class Book{
    // Kotlin 注解也可通过属性名来指定属性值
    @KotlinTag(name="kotlin", age=14)
    // Java 注解只能通过属性名来指定属性值
    @JavaTag(name="java", age=28)
    fun test(){
    }
}
```

从上面程序的前两行粗体字代码可以看出，Kotlin 注解既支持使用位置来指定属性值（第一个值传给第一个属性，第二个值传给第二个属性，依此类推），也支持使用属性名来指定属性值（这是传统 Java 注解的使用方式）；而 Java 注解只能通过属性名来指定属性值。

如果 Java 注解中的 value 属性是数组类型，那么它会变成 Kotlin 注解的 vararg 属性，因此直接为它传入多个属性值即可。例如如下 Java 注解。

程序清单：codes\11\11.4\JavaTagWithArray.java

```
public @interface JavaTagWithArray{
    public String[] value();
}
```

数组类型的 value 属性会变成 Kotlin 注解的 vararg 属性，因此可以在 Kotlin 程序中按如下方式使用该注解。

程序清单：codes\11\11.4\UseJavaTagWithArray.kt

```
// 直接传入多个属性值
@JavaTagWithArray("java", "kotlin", "go")
```

```
class Stock
```

但如果其他名称的属性是数组类型，那么在 Kotlin 中使用该注解时必须显式使用 arrayOf() 函数来构建数组。例如如下 Java 注解。

程序清单：codes\11\11.4\JavaTagWithArray2.java

```
public @interface JavaTagWithArray2{
    public String[] infos();
}
```

上面注解的 infos 属性是数组类型，因此在 Kotlin 中使用该注解时必须显式使用 arrayOf() 函数来构建数组。例如如下代码。

程序清单：codes\11\11.4\UseJavaTagWithArray2.kt

```
// 对于名称不是 value 的属性，需要显式使用 arrayOf()函数来构建数组
@JavaTagWithArray2(infos=arrayOf("java", "kotlin", "go"))
class Stock
```

11.5 本章小结

本章主要介绍了 Kotlin 的注解支持，通过使用注解可以为程序提供一些元数据（通常是一些配置信息，很多时候注解用于取代原来的 XML 配置），这些配置数据可以在编译、运行时被读取，从而提供更多额外的处理信息。本章详细介绍了 Kotlin 定义注解的基本语法，以及带属性、构造器的注解。此外，本章也详细讲解了 Kotlin 提供的 4 个用于修饰注解的元注解的用法。本章还介绍了通过 Kotlin API 在运行时读取注解、使用注解的方法。本章最后介绍了 Kotlin 注解与 Java 程序的兼容性处理，包括指定注解的作用目标，以及如何使用 Java 注解等。

CHAPTER

12

第12章 Kotlin与Java互相调用

本章要点

- Kotlin 调用 Java 的注意点
- Java 调用 Kotlin 的注意点
- 泛型型变、checked 异常、属性的差异
- 获取 KClass 对象
- 通过 KClass 对象获取类的详细信息
- 通过反射创建对象
- 获取构造器引用
- 通过反射调用方法
- 获取函数或方法引用
- 通过反射操作属性
- 获取属性引用
- 绑定的属性和方法引用

本章作为本书的最后一章，将会介绍 Java 与 Kotlin 的相互调用。由于 Kotlin 最初设计就是兼容 JVM 平台的，因此 Java 与 Kotlin 相互可以平滑调用。但两种语言之间还是存在一些细微差异的，因此相互调用时需要注意这些差异点。本章将会详细介绍这些差异点。

本章将重点介绍 kotlin.reflect、kotlin.reflect.full、kotlin.reflect.jvm 包下的接口和类，包括 KClass、KCallable、KFunction、KProperty 及其子接口等，它们分别代表类、函数（包括构造器、方法）、属性。Kotlin 程序可以使用这些接口动态地获取某个对象、某个类的运行时信息，并可以动态地创建对象，调用方法，访问并修改指定对象的属性值。

12.1 Kotlin 调用 Java

由于 Kotlin 本身并没有提供强大的类库支持，Kotlin 只是一种语言，因此 Kotlin 调用 Java 通常都是自然而然的事情。正如我们在前面程序中所看到的，在 Kotlin 中调用 Java 的 Date、JFrame 等类完全没有任何问题。

Kotlin 调用 Java 时只是有一些小小的注意点，下面对这些内容做一下简单的说明。

12.1.1 属性

Kotlin 调用属性实际上就是访问 getter、setter 方法，因此 Java 类只要提供了 getter 方法，Kotlin 就可将其当成只读属性；如果 Java 类提供了 getter、setter 方法，Kotlin 就可将其当成读写属性。



提示：

如果 getter 方法的返回值类型是 boolean，该 getter 方法名也以 is 开头，此时 Kotlin 会将其当成属性名与 getter 方法同名的属性。比如 boolean isMarried()方法可作为 Kotlin 的只读属性，该属性名为 isMarried。

下面程序定义了一个简单的 Java 类。

程序清单：codes\12\12.1\User.java

```
public class User{
    private String name;
    // name 的 setter 和 getter 方法
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return this.name;
    }
    // married 的 setter 和 getter 方法
    public void setMarried(boolean married){
        System.out.println("调用 setMarried 方法，参数为：" + married);
    }
    public boolean isMarried(){
        return true;
    }
    public int getAge(){
        return new java.util.Random().nextInt(100);
    }
}
```

上面 User 类定义了一个 name 成员变量，并提供了 setName()和 getName()两个方法，因此 Kotlin 可将其当成名为 name 的读写属性；User 类还定义了 setMarried()和 isMarried()两个方法，因此 Kotlin 可将其当成名为 isMarried 的读写属性；User 类还定义了 getAge()方法，因此 Kotlin 可将其当成名为 age 的只读属性。

注意：

其实 Java 是否包含成员变量不重要，关键是 getter、setter 方法：如果只有 getter 方法，那么就是只读属性；如果有 getter、setter 方法，那么就是读写属性。



如下程序示范了 User 类的测试程序。

程序清单：codes\12\12.1\UserTest.kt

```
fun main(args: Array<String>) {
    val user = User()
    // 操作 name 读写属性
    user.name = "yeeku"
    println(user.name)
    // 操作 isMarried 读写属性
    user.isMarried = true
    println(user.isMarried)
    // 读取 age 只读属性
    println(user.age)
}
```

编译、运行上面程序，可看到如下输出：

```
yeeku
调用 setMarried 方法，参数为：true
true
17
```

从上面的输出结果可以看出，对属性赋值就是调用 setter 方法；访问属性的值就是调用 getter 方法。

12.1.2 void 和调用名为关键字的成员

如果 Java 方法的返回值是 void，在 Kotlin 中则对应于 Unit 返回值类型。

由于 Kotlin 的关键字比 Java 多（比如 Kotlin 的 is、object、in 在 Java 语言中都不是关键字），因此可能出现一种情况：Java 的类名、接口名、方法名等是 Kotlin 的关键字。此时就需要使用反引号（就是键盘上数字 1 左边的那个键）对关键字进行转义。

例如，如下 Java 类定义了一个名为 in 的方法。

程序清单：codes\12\12.1\InMethod.java

```
public class InMethod{
    public void in(){
        System.out.println("in 方法");
    }
}
```

为了在 Kotlin 中调用该方法，程序就需要对该方法名进行转义。

程序清单：codes\12\12.1\InMethodTest.kt

```
fun main(args: Array<String>) {
```

```
    val im = InMethod()
//    im.in() // in是关键字，报错
    // 调用in()方法
    im.`in`() // 对in关键字转义
}
```

12.1.3 Kotlin的已映射类型

虽然Kotlin并未完整地提供整套类库，但Kotlin还是为部分Java类提供了特殊处理，这部分Java类被映射到Kotlin类，这种映射只在编译阶段发生，在运行阶段依然使用Java类型。

表12.1显示了Java基本类型与Kotlin类之间的映射关系。

表12.1 Java基本类型与Kotlin类之间的映射关系

Java基本类型	Kotlin类
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

Java基本类型的包装类则映射到Kotlin类的可空类型，如表12.2所示。

表12.2 Java基本类型的包装类与Kotlin类之间的映射关系

基本类型的包装类	Kotlin类
java.lang.Byte	kotlin.Byte?
java.lang.Short	kotlin.Short?
java.lang.Integer	kotlin.Int?
java.lang.Long	kotlin.Long?
java.lang.Character	kotlin.Char?
java.lang.Float	kotlin.Float?
java.lang.Double	kotlin.Double?
java.lang.Boolean	kotlin.Boolean?

此外，Kotlin还为Java的一些常用类提供了映射，如表12.3所示。

表12.3 Java的常用类与Kotlin类之间的映射关系

Java的常用类	Kotlin类
java.lang.Object	kotlin.Any!
java.lang.Cloneable	kotlin.Cloneable
java.lang.Comparable	kotlin.Comparable!
java.lang.Enum	kotlin.Enum!
java.lang.Annotation	kotlin.Annotation!
java.lang.Deprected	kotlin.Deprected!
java.lang.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

12.1.4 Kotlin 对 Java 泛型的转换

Kotlin 不支持 Java 的泛型通配符语法，但 Kotlin 提供了使用处型变来代替泛型通配符。因此 Java 的通配符将会转换为 Kotlin 的使用处型变（类型投影）。此外，Java 的原始类型则转换为 Kotlin 的星号投影。它们的关系如表 12.4 所示。

表 12.4 Java 泛型和 Kotlin 泛型的转换关系

Java 泛型	Kotlin 语法
Foo<? extends Bar>	Foo<out Bar!>!
Foo<? super Bar>	Foo<in Bar!>!
Foo<*>	Foo<out Any?>!
Foo	Foo<*>

Kotlin 和 Java 一样，它们都无法在运行时保留泛型信息，因此在 Kotlin 中使用 is 运算符进行类型检测时只能检测星号投影（相当于 Java 的原始类型），不能检测泛型信息。例如如下代码。

程序清单：codes\12\12.1\IsNoGeneric.kt

```
fun main(args: Array<String>) {
    val list = listOf(2, 3, 10)
    // is 判断不能检测泛型参数
//    println(list is List<String>)
    // 只能检测星号投影
    println(list is List<*>) // 输出 true
}
```

正如从上面程序中所看到的，程序检测 list is List<String>是无效的，这是因为程序在运行时只能判断是否为 List，无法判断它的泛型信息。

12.1.5 对 Java 数组的处理

相比 Java 数组，Kotlin 数组有一个重大的改变：Kotlin 数组是不型变的。因此 Array<String>不能赋值给 Array<Object>变量。但 Java 数组不同，Java 数组是型变的，因此 String[]可以直接赋值给 Object[]变量。



提示：

Java 数组的型变可以说是一个缺陷，这个缺陷容易导致程序运行时出现异常。例如，如下代码编译完全没有问题，但运行时就会引发 ArrayStoreException 异常。

```
String[] strArr = new String[10];
Object[] objArr = strArr;
objArr[0] = 2;
```

此外，Java 还支持 int[]、long[]等基本类型的数组，这种数组可以避免拆箱、装箱带来的性能开销，但 Kotlin 的 Array 类不支持这种基本类型的数组。为此，Kotlin 提供了 ByteArray、ShortArray、IntArray、LongArray、CharArray、FloatArray、DoubleArray、BooleanArray 数组，这几种数组专门用于代替 Java 的 byte[]、short[]等基本类型的数组。比如定义了如下 Java 方法。

程序清单：codes\12\12.1\IntArrayFun.java

```
public class IntArrayFun{
    public int sum(int[] arr){
```

```
        int sum = 0;
        for (int i = 0; i < arr.length ; i++){
            sum += arr[i];
        }
        return sum;
    }
}
```

对于上面的sum()方法，它需要的参数是int[]类型，因此Kotlin程序不能用Array<Int>数组来调用该方法，而应该使用IntArray数组来调用该方法。例如如下程序。

程序清单：codes\12\12.1\IntArrayFunTest.kt

```
fun main(args: Array<String>) {
    val iaf = IntArrayFun()
    // 由于sum()方法需要的参数是int[]类型，因此此处需要使用IntArray对象
    val intArr = intArrayOf(2, 4, 10)
    println(iaf.sum(intArr))
}
```

此外，读者无须对Kotlin数组访问效率有任何担心，Kotlin编译器会对数组访问进行优化，它会采用和Java相似的方式来访问数组元素（Java访问数组元素的方式是：根据索引直接计算数组元素的内存地址，访问效率非常高），因此Kotlin在访问元素时没有性能损耗。

例如如下程序。

程序清单：codes\12\12.1\ArrayTest.kt

```
fun main(args: Array<String>) {
    val array = arrayOf(1, 2, 3, 4)
    // 不会使用get()、set()方法访问数组元素
    // 实际上还是通过Java数组的快速访问
    array[2] = array[2] * 2
    // 使用索引定位是基于Java数组元素所在内存地址的快速访问
    // 不需要创建迭代器
    for (i in array.indices) {
        array[i] += 2
    }
    // for-in循环也是基于Java数组元素所在内存地址的快速访问
    // 不需要创建迭代器
    for (x in array) {
        println(x)
    }
}
```

正如从上面程序中所看到的，不管程序采用哪种方式来遍历数组的元素，Kotlin总会将其优化成和Java类似的访问数组元素的方式，因此开发者无须担心性能问题。

12.1.6 调用参数个数可变的方法

对于参数个数可变的方法，Java可以直接传入一个数组，但Kotlin不行。Kotlin要求只能传入多个参数值，但也可通过使用“*”解开数组的方式来传入多个数组元素作为参数值。

例如，如下程序定义了一个Java方法。

程序清单：codes\12\12.1\JavaVarargs.java

```
public class JavaVarargs {
    // 定义参数个数可变的方法
    public void test(int... nums){
```

```
        for (int i = 0; i < nums.length ; i++ ){
            System.out.println(nums[i]);
        }
    }
}
```

对于上面的 test()方法，程序可直接传入多个 int 值来调用该方法；但如果程序已有一个 IntArray 数组，则需要使用“*”将数组解开成多个数组元素传入。例如如下程序。

程序清单：codes\12\12.1\JavaVarargsTest.kt

```
fun main(args: Array<String>) {
    val jv = JavaVarargs()
    jv.test(2, 4, 5)
    val intArr = intArrayOf(6, 20, 33)
    // 将数组解开成多个元素
    jv.test(*intArr)
}
```

12.1.7 checked 异常

由于 Kotlin 没有 checked 异常，因此对于 Java 中可能引发 checked 异常的方法、构造器，Kotlin 则不会引发该异常，Kotlin 既可捕获该异常，也可完全不理会该异常（无须使用 throws 声明抛出异常）。例如如下程序。

程序清单：codes\12\12.1\CheckedExceptionTest.kt

```
import java.io.*
fun main(args: Array<String>) {
    // 下面代码可能引发 IOException (checked 异常)
    // 但 Kotlin 并不强制处理该异常
    val fis = FileInputStream("aa.txt")
    println(fis.read())
}
```

12.1.8 Object 的处理

Java 的 java.lang.Object 对应于 Kotlin 中的 Any，又因为 Any 只声明了 toString()、hashCode()和 equals()方法，因此需要考虑如何使用 java.lang.Object 类中其他方法的问题。

1. wait()/notify()/notifyAll()

这三个方法都是 Java 程序中同步监视器支持调用的方法，一般来说，只有在线程通信中才会用到，而且 Java 提供了更好的 Condition 来控制线程通信，因此一般不需要调用这三个方法。

但如果编程时真的需要让同步监视器调用这三个方法，则可以在程序中将 Any 转型为 java.lang.Object，然后再调用这三个方法。例如如下代码：

```
(foo as java.lang.Object).wait()
```

2. getClass()

java.lang.Object 对象的 getClass()方法用于获取该对象的 Java 类，Kotlin 的对象则有两种方式来获取该对象的 Java 类。例如如下代码：

```
// 获取 obj 对象的 Java 类
obj::class.java
```

```
obj.javaClass
```

3. clone()

如果要让对象重写 clone()方法，则需要让该类实现 kotlin.Cloneable 接口。例如如下代码：

```
class Example : Cloneable {
    override fun clone(): Any { …… }
}
```

4. finalize()

java.lang.Object 的 finalize()方法主要用于完成一些资源清理的工作，GC 在回收某个对象之前，JVM 会自动调用该对象的 finalize()方法。如果要重写 finalize()方法，则只要在类中实现该方法即可，不需要使用 override 关键字（因为在 Any 类中并没有声明 finalize()方法）。例如如下代码：

```
class Foo {
    protected fun finalize() {
        // 实现资源清理的逻辑
    }
}
```

根据 Java 的规则，finalize()方法不能定义成 private。

12.1.9 访问静态成员

虽然 Kotlin 本身没有提供 static 关键字，但 Kotlin 提供了伴生对象来实现静态成员，因此 Java 类中的静态成员都可通过伴生对象的语法来调用。比如如下代码。

程序清单：codes\12\12.1\StaticMemberTest.kt

```
fun main(args: Array<String>) {
    // 调用 Runtime 的静态方法（就像调用伴生对象的方法一样）
    val rt = Runtime.getRuntime()
    println(rt)
    // 访问 java.awt.BorderLayout 的 NORTH 静态成员（就像访问伴生对象的属性一样）
    val str = java.awt.BorderLayout.NORTH
    println(str)
}
```

12.1.10 SAM 转换

Java 8 支持使用 Lambda 表达式来作为函数式接口的实例（这种机制被称为 SAM 转换），Kotlin 同样支持这个功能。比如如下代码。

程序清单：codes\12\12.1\SamTest.kt

```
import java.util.function.*
import java.util.concurrent.*
fun main(args: Array<String>) {
    // 使用 Lambda 表达式来创建函数式接口（Predicate）的对象
    val pred = Predicate<Int>{t -> t > 5}
    val list = arrayListOf(2, 200, 10, 34, 30, 44)
    // 使用 pred 对 List 集合进行过滤
    list.removeIf(pred)
    println(list) // 输出[2]
    // 使用 Lambda 表达式来创建函数式接口（Runnable）的对象
    val rn = Runnable{
```

```
        for (i in 0..10) {
            println(i)
        }
    }
    // 通过 Runnable 对象创建、启动线程
    Thread(rn).start()
    val executor = ThreadPoolExecutor()
    // 由于 executor 的 execute()方法需要一个 Runnable 对象
    // 因此程序可自动将符合该接口规范的 Lambda 表达式创建成 Runnable 对象
    executor.execute { println("This runs in a thread pool") }
    // 当然也可在方法中显式指定 Lambda 表达式创建的是 Runnable 对象
    executor.execute(Runnable { println("This runs in a thread pool") })
}
```

与 Java 类似，Lambda 表达式只能创建函数式接口的对象，不能创建抽象类的对象，即使该抽象类只有一个抽象方法也不行；同理，Kotlin 的 Lambda 表达式也只能创建函数式接口的对象，不能创建抽象类的对象。

12.1.11 在 Kotlin 中使用 JNI

如果要在 Java 中使用 JNI，则应该使用 native 修饰该方法，表明该方法将会交给平台本地的 C 或 C++代码来实现。

Kotlin 同样支持该机制，只不过 Kotlin 不使用 native 关键字，而是使用 external 关键字。例如如下函数：

```
external fun foo(x: Int): Double
```

该函数使用了 external 修饰，因此该函数不能指定函数体。该函数的函数体将会由平台本地的 C 或 C++代码来实现。



提示：

Kotlin 的 external 就相当于 Java 的 native。

12.2 Java 调用 Kotlin

由于 Kotlin 程序编译之后本身就是完全兼容 JVM 规范的，因此 Java 调用 Kotlin 也是非常方便的。只是有一些小的注意点需要跟读者交代一下。

12.2.1 属性

前面在介绍 Kotlin 属性时已经讲过，Kotlin 属性可编译成如下三个成员。

- 一个 private 实例变量，实例变量名与属性名相同（如果该属性具有幕后字段）。
- 一个 getter 方法，方法名为属性名添加 get 前缀。
- 一个 setter 方法，方法名为属性名添加 set 前缀（读写属性才有 setter 方法）。

例如如下属性：

```
var name: String
```

该属性将会被 Kotlin 编译成如下三个成员：

```
private String name;
public String getName() {
```

```
    return name;
}
public void setName(String name) {
    this. name = name;
}
```

但如果属性名以 is 开头（属性类型并不要求一定是 Boolean 类型），那么该属性对应的生成规则略有差别。

- 实例变量名、getter 方法名与属性名相同。
- setter 方法名为属性名去掉 is 前缀，添加 set 前缀。

例如如下属性：

```
var isMarried: Boolean
```

该属性将会被 Kotlin 编译成如下三个成员。

```
private boolean isMarried;
public boolean isMarried() {
    return isMarried;
}
public void setMarried(boolean isMarried) {
    this. isMarried = isMarried;
}
```

12.2.2 包级函数

所谓包级函数，是指在 Kotlin 中直接定义的顶级函数（不在任何类中定义）。但实际上 Kotlin 编译器会为整个 Kotlin 源文件生成一个类（只要该源文件中包含了顶级函数、顶级变量），而这些顶级函数、顶级变量都会变成该类的静态方法、静态变量。例如如下代码。

程序清单：codes\12\12.2\TopLevel.kt

```
var name: String = ""
fun test(){
}
```

由于上面源文件中包含了顶级成员（顶级函数或变量），在默认情况下，Kotlin 编译器会为该源文件生成一个 TopLevelKt.class 文件，这些顶级成员就变成该 TopLevelKt 的静态成员。例如，上面两个顶级成员变成如下静态成员：

```
public class TopLevelKt{
    private static java.lang.String name;
    public static final java.lang.String getName(){...}
    public static final void setName(java.lang.String){...}

    public static final void test(){...}
}
```

理解了 Kotlin 的顶级成员编译之后的形式，读者自然也就知道如何调用它们了。

在默认情况下，Kotlin 为包含顶级成员的源文件所生成类的类名总是文件名+Kt 后缀；但 Kotlin 也允许使用@JvmName 注解（该注解用来修饰文件本身，它是一个 AnnotationRetention.SOURCE 的注解）来改变 Kotlin 编译生成的类名。

例如如下程序。

程序清单：codes\12\12.2\JvmName.kt

```
// 指定生成的类
```

```
@file:JvmName("FkUtils")
package org.crazyit
fun test() {
    println("FkUtils 的 test()方法")
}
```

上面程序中粗体字代码指定该源文件所生成的类名为 FkUtils，因此编译该源文件将会生成一个 FkUtils.class 文件。

Java 程序可按如下方式来调用该 test()方法。

程序清单：codes\12\12.2\JvmNameTest.java

```
import org.crazyit.FkUtils;
public class JvmNameTest{
    public static void main(String[] args){
        // 调用 FkUtils 的 test()类方法
        FkUtils.test();
    }
}
```

从上面的粗体字代码可以看出，此时 Kotlin 源文件对应的类名变成了 FkUtils。

还有一种极端情况，就是多个 Kotlin 源文件要生成相同的 Java 类（包名相同且类名相同，或指定了相同的@JvmName 注解），这种情况默认会引发错误。但 Kotlin 非常智能，我们可以指定 Kotlin 为这些 Kotlin 源文件统一生成一个 Java 类，而该 Java 类将会包含不同源文件中的顶级成员。为了实现这种效果，需要使用@JvmMultifileClass 注解。

例如，如下两个 Kotlin 程序使用了@JvmMultifileClass 注解。

程序清单：codes\12\12.2\JvmMultifileClass1.kt

```
// 指定生成的类
@file:JvmName("CiUtils")
// 指定为多个文件中的顶级成员统一生成一个类
@file:JvmMultifileClass
package org.crazyit
fun foo() {
    println("CiUtils 的 foo()方法")
}
```

程序清单：codes\12\12.2\JvmMultifileClass2.kt

```
// 指定生成的类
@file:JvmName("CiUtils")
// 指定为多个文件中的顶级成员统一生成一个类
@file:JvmMultifileClass
package org.crazyit
fun bar() {
    println("CiUtils 的 bar()方法")
}
```

上面两个 Kotlin 源程序都需要生成 org.crazyit.CiUtils 类，在默认情况下这将导致错误。因此，上面程序使用了@JvmMultifileClass 注解，这就会告诉 Kotlin 编译器将两个文件合并到一个类中。

如果读者打算使用 kotlinc 编译器编译上面两个程序，则不能分开编译两个文件，要同时编译上面两个文件，才可看到两个 Kotlin 源文件生成一个 class 文件。比如使用如下命令来编译：

```
kotlinc -d . -cp . JvmMultifileClass1.kt JvmMultifileClass2.kt
```

执行上面命令，将看到生成一个 org.crazyit.CiUtils 类。接下来 Java 程序可采用如下方式来调用该类中的 foo()、bar()方法。

程序清单：codes\12\12.2\JvmMultifileClassTest.java

```
import org.crazyit.*;
public class JvmMultifileClassTest{
    public static void main(String[] args){
        CiUtils.foo();
        CiUtils.bar();
    }
}
```

12.2.3 实例变量

Kotlin 允许将属性暴露成实例变量，只要在程序中使用@JvmField 修饰该属性即可，暴露出来的属性将会和 Kotlin 属性具有相同的访问权限。

使用@JvmField 将属性暴露成实例变量的要求如下。

- 该属性具有幕后字段。
- 该属性必须是非 private 访问控制的。
- 该属性不能使用 open、override、const 修饰。
- 该属性不能是委托属性。

例如如下 Kotlin 程序。

程序清单：codes\12\12.2\InstanceField.kt

```
class InstanceField(name: String){
    @JvmField val myName = name
}
```

上面粗体字代码定义了一个属性，并使用@JvmField 注解修饰了该属性，这样系统就会把该属性的幕后字段暴露成实例变量（默认是 public 访问权限）。

下面 Java 程序示范了访问该类的对象的实例变量。

程序清单：codes\12\12.2\InstanceFieldTest.java

```
public class InstanceFieldTest{
    public static void main(String[] args){
        InstanceField ins = new InstanceField("Kotlin");
        // 访问 InstanceField 对象的实例变量
        System.out.println(ins.myName);
    }
}
```

12.2.4 类变量

在命名对象（对象声明）或伴生对象中声明的属性会在该命名对象或包含伴生对象的类中具有静态幕后字段（类变量）。但这些类变量通常是 private 访问权限，程序可通过如下三种方式之一将它们暴露出来。

- 使用@JvmField 注解修饰。
- 使用 lateinit 修饰。
- 使用 const 修饰。

例如，如下程序在 MyClass 类中定义了一个伴生对象。

程序清单：codes\12\12.2\MyClass.kt

```
class MyClass{
    // 使用 companion 修饰的伴生对象
    companion object MyObject1{
        @JvmField val name = "name 属性值"
    }
}
```

上面粗体字代码在伴生对象中定义了一个属性。根据前面的介绍我们知道：伴生对象本来就是用来弥补 Kotlin 没有 static 关键字的不足的，因此伴生对象中的属性实际上就相当于 MyClass 的类变量，但它默认是 private 访问权限。上面粗体字代码使用了@JvmField 修饰，这样该类变量就变成了与该属性具有相同的访问控制符：public。

因此，下面 Java 程序可直接访问 MyClass 的 name 类变量。

程序清单：codes\12\12.2\MyClassTest.java

```
public class MyClassTest{
    public static void main(String[] args){
        // 访问 MyClass 的 name 类变量
        System.out.println(MyClass.name);
    }
}
```

在命名对象或伴生对象中的延迟初始化属性（lateinit）具有与该属性的 setter 方法相同的访问控制符的类变量。道理很简单：Java 并不支持命名对象，因此 Kotlin 的命名对象在 Java 中其实表现为一个单例类，命名对象的属性就变成了类变量；又由于 lateinit 属性需要等到使用时才赋值，因此 Kotlin 必须将该类变量暴露出来——否则 Java 程序将无法对该类变量赋值。

例如如下 Kotlin 程序。

程序清单：codes\12\12.2\MyClass.kt

```
// 定义命名对象
object MyObject {
    // 定义延迟初始化属性
    lateinit var name: String
}
```

上面程序在 MyObject 命名对象中定义了延迟初始化属性，该属性将会被暴露成类变量，因此 Java 程序可按如下方式来调用它。

程序清单：codes\12\12.2\MyClassTest.java

```
public class MyObjectTest{
    public static void main(String[] args){
        // 直接访问 MyObject 类的 name 类变量
        MyObject.name = "Kotlin";
        System.out.println(MyObject.name);
    }
}
```

此外，在 Kotlin 程序中使用 const 修饰的属性，不管是在顶层定义的属性，还是在对象中定义的属性，只要使用了 const 修饰，它就会变成有 public static final 修饰的类变量。



提示：

不知读者是否还记得《疯狂 Java 讲义》中介绍过的“宏替换变量”？“宏替换变量”实际上就是此处用 const 修饰的变量，因此用 const 修饰的变量总会变成有 public static final 修饰。

例如如下 Kotlin 程序。

程序清单：codes\12\12.2\Const.kt

```
const val MAX = 239
object Obj {
    const val NAME = "疯狂软件"
}
class Css {
    companion object {
        // 使用 const 修饰的变量
        const val SITE = "www.fkit.org"
    }
}
```

上面程序中定义了一个顶层的 const 属性，还在命名对象中定义了一个 const 属性，在伴生对象中定义了一个 const 属性，这些 const 属性都会变成有 public static final 修饰的属性。因此可用如下 Java 程序来调用它们。

程序清单：codes\12\12.2\ConstTest.java

```
public class ConstTest{
    public static void main(String[] args){
        System.out.println(ConstKt.MAX);
        System.out.println(Obj.NAME);
        System.out.println(Css.SITE);
    }
}
```

12.2.5 类方法

可以肯定的是，Kotlin 的顶级函数会被转换成类方法。

此外，Kotlin 还可以将命名对象或伴生对象中定义的方法转换成类方法——如果这些方法使用@ JvmStatic 修饰的话。一旦使用该注解，编译器就既会在相应对象的类中生成类方法，也会在对象自身中生成实例方法。

例如如下 Kotlin 程序。

程序清单：codes\12\12.2\StaticMethod.kt

```
// 指定零个父类型的命名对象
object MyObject {
    // 方法
    fun test(){
        println("test 方法")
    }
    @JvmStatic fun foo(){
        println("有@JvmStatic 修饰的 foo()方法")
    }
}
class MyClass {
```

```
    // 省略名字的伴生对象
    companion object{
        // 方法
        fun test(){
            println("test 方法")
        }
        @JvmStatic fun output(msg: String){
            for(i in 1..6){
                println("<h${i}>${msg}</h${i}>")
            }
        }
    }
}
```

上面程序中定义了一个命名对象和一个伴生对象，其中命名对象中包含了两个方法：一个方法没有@JvmStatic 修饰，另一个方法有@JvmStatic 修饰；伴生对象中也包含了两个方法：一个方法没有@JvmStatic 修饰，另一个方法有@JvmStatic 修饰。

这样有@JvmStatic 修饰的方法既会在相应对象的类中生成类方法，也会在对象自身中生成实例方法。例如，可使用如下 Java 代码来调用这些方法。

程序清单：codes\12\12.2\StaticMethodTest.java

```
public class StaticMethodTest{
    public static void main(String[] args){
        // 通过 INSTANCE 访问 MyObject 的单例，通过单例访问 test()方法
        MyObject.INSTANCE.test();
        // 错误，test()方法只是单例对象的实例方法
//        MyObject.test();
        // 通过 INSTANCE 访问 MyObject 的单例，通过单例访问 foo()方法
        MyObject.INSTANCE.foo();
        // 正确，foo()方法有@JvmStatic 修饰，因此也是 MyObject 类的类方法
        MyObject.foo();
        // 通过 Companion 访问 MyClass 的伴生对象，通过伴生对象访问 test()方法
        MyClass.Companion.test();
        // 错误，test()方法只是伴生对象的实例方法
//        MyClass.test();
        // 通过 Companion 访问 MyClass 的伴生对象，通过伴生对象访问 output()方法
        MyClass.Companion.output("疯狂软件");
        // 正确，output()方法有@JvmStatic 修饰，因此也是 MyClass 类的类方法
        MyClass.output("疯狂软件");
    }
}
```

从上面程序中可以看出，对于命名对象中有@JvmStatic 修饰的方法，既可通过命名对象的 INSTANCE（引用单例对象）来调用该方法，也可通过命名对象对应的类来调用该方法；对于伴生对象中有@JvmStatic 修饰的方法，既可通过伴生对象所在类的 Companion（引用伴生对象）来调用该方法，也可通过伴生对象所在的类来调用该方法。

12.2.6 访问控制符的对应关系

Kotlin 的访问控制符与 Java 的对应关系如下。

- **private 成员**：依然编译成 Java 的 private 成员。
- **protected 成员**：依然编译成 Java 的 protected 成员。但需要注意的是，在 Java 程序中，protected 成员可以被同一个包中的其他成员访问，但 Kotlin 不行（Kotlin 中的 protected

成员只能被当前类及其子类成员访问）。

- **internal 成员**：会编译成 Java 的 public 成员。但编译成 Java 类时会通过名字修饰来避免在 Java 中被意外使用到。
- **public 成员**：依然编译成 Java 的 public 成员。

12.2.7 获取 KClass

前面介绍过使用 Kotlin 来获取 Java 的 Class 对象；反过来，有时候 Java 也需要获取 Kotlin 的 KClass 对象。为了在 Java 中获取 Kotlin 的 KClass 对象，必须通过调用 Class<T>.kotlin 扩展属性的等价形式来实现。

例如如下程序。

程序清单：codes\12\12.2\KClassTest.java

```
import java.util.ArrayList;
import kotlin.reflect.*;
public class KClassTest{
    public static void main(String[] args)throws Exception{
        Class<?> clazz = Class.forName("java.util.ArrayList");
        System.out.println(clazz);
        // 获取 Class 对象的 KClass 对象
        KClass kc = kotlin.jvm.JvmClassMappingKt.getKotlinClass(clazz);
        System.out.println(kc);
    }
}
```

上面粗体字代码用于获取指定 Class 对象的 KClass 对象。

12.2.8 使用@JvmName 解决签名冲突

有些时候，在 Kotlin 中要定义两个同名函数，但 JVM 平台无法区分这两个函数。典型的情况就是类型擦除引起的问题。例如如下两个函数。

程序清单：codes\12\12.2\NameConflict.kt

```
fun List<String>.filterValid(): List<String>{
    val result = mutableListOf<String>()
    for (s in this) {
        if (s.length < 5) {
            result.add(s)
        }
    }
    return result.toList()
}
@JvmName("fileterValidInt")
fun List<Int>.filterValid(): List<Int>{
    val result = mutableListOf<Int>()
    for (i in this) {
        if (i < 20) {
            result.add(i)
        }
    }
    return result.toList()
}
```

上面程序中定义了两个扩展方法，虽然程序看上去是分别为 List<String>、List<Int>扩展

的方法，但由于编译器编译之后会产生类型擦除，因此上面两个方法的签名都是List.filterValid():List。所以在 JVM 平台上无法区分这两个方法。

为了让JVM平台能区分这两个方法，程序中粗体字代码使用了@JvmName("fileterValidInt")来标注后一个方法，因此后一个方法在 JVM 平台上会编译成 fileterValidInt()方法。

可使用如下 Java 程序来调用这两个方法。

程序清单：codes\12\12.2\NameConflictTest.java

```
import java.util.*;
public class NameConflictTest{
    public static void main(String[] args){
        List<String> list1 = Arrays.asList("Java",
            "Kotlin", "Erlang", "Go", "Swift");
        // 调用 List<String>扩展的 filterValid()的方法
        System.out.println(NameConflictKt.filterValid(list1));
        List<Integer> list2 = Arrays.asList(200,
            34, 12, 35, -23);
        // 调用 List<Int>扩展的 fileterValid()（在字节码中名为 fileterValidInt）方法
        System.out.println(NameConflictKt.fileterValidInt(list2));
    }
}
```

从上面程序中可以看出，增加@JvmName 注解之后，程序就可以正常区分两个签名相同的方法了。

@JvmName 注解也适用于属性 x 和函数 getX()共存。例如如下代码：

```
val x: Int
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

12.2.9 生成重载

我们知道，Kotlin 为方法（或函数）提供了参数默认值来避免函数重载过多的问题。但对于这种参数有默认值的方法（或函数），编译器默认只生成一个方法：默认带所有参数的方法。

为了让编译器能为带参数默认值的方法（或函数）生成多个重载的方法（或函数），可考虑使用@JvmOverloads 注解。

需要说明的是，@JvmOverloads 注解也适用于构造器、静态方法等。它不适用于抽象方法，包括在接口中定义的方法。

如下程序示范了使用@JvmOverloads 注解让编译器为带参数默认值的方法生成多个重载的方法。

程序清单：codes\12\12.2\Overload.kt

```
@JvmOverloads
fun test(name: String , len: Int = 20, price: Double = 2.2) {
    println(name)
    println(len)
    println(price)
}
```

上面定义了一个带两个默认参数的函数，并使用@JvmOverloads 修饰该函数，这样编译器将会为该函数生成如下三个方法。

```
public static final void test(java.lang.String, int, doube)
public static final void test(java.lang.String, int)
public static final void test(java.lang.String)
```

因此，可用如下 Java 程序来调用这些方法。

程序清单：codes\12\12.2\OverloadTest.java

```
public class OverloadTest{
    public static void main(String[] args){
        OverloadKt.test("yeeku", 182, 3.4);
        OverloadKt.test("yeeku", 182);
        OverloadKt.test("yeeku");
    }
}
```

有一点需要指出：如果一个类的所有构造器的参数都有默认值，这样就能以构造参数的默认值来调用构造器（就像调用无参数的构造器），因此在 JVM 平台上相当于生成一个 public 的无参数构造器，这个功能不需要@JvmOverloads 注解。

12.2.10 checked 异常

前面已经介绍过，Kotlin 没有 checked 异常，因此 Kotlin 也无须使用 throws 抛出异常。这样即使 Kotlin 的方法（或函数）本身可能抛出 checked 异常，Java 调用该方法（或函数）时编译器也不会检查该异常。

例如，如下函数就存在这个问题。

```
fun foo() {
    val fis = FileInputStream("a.txt")
}
```

如果我们希望 Java 调用该函数时编译器会检查该 IOException，则可使用@Throws 注解修饰该函数。例如，将上面代码改为如下形式。

程序清单：codes\12\12.2\CheckedException.kt

```
import java.io.*
@Throws(IOException::class)
fun foo() {
    val fis = FileInputStream("a.txt")
}
```

这样上面 foo()函数就相当于声明了 throws IOException，因此 Java 程序调用该函数时编译器会检查该 checked 异常，程序要么捕获该异常，要么显式声明抛出该异常。例如如下 Java 程序。

程序清单：codes\12\12.2\CheckedExceptionTest.java

```
public class CheckedExceptionTest {
    public static void main(String[] args){
        // 下面代码错误，编译器会检查 IOException
        // 因此程序要么捕获该异常，要么显式声明抛出该异常
        CheckedExceptionKt.foo();
    }
}
```

12.2.11 泛型的型变

首先回顾一下 Kotlin 泛型型变和 Java 泛型型变的区别：Kotlin 的泛型支持声明处型变，

而 Java 的泛型只支持使用处型变（通过通配符形式，可以支持协变和逆变两种形式）。因此，对于 Kotlin 的声明处型变，必须转换成 Java 的使用处型变。其转换规则是：

- 对于协变类型的泛型类 Bar<out T>，当它作为参数出现时，Kotlin 会自动将 Bar<Base>类型的参数替换成 Bar<? extends Base>。
- 对于逆变类型的泛型类 Foo<in T>，当它作为参数出现时，Kotlin 会自动将 Foo<Sub>类型的参数替换成 Foo<? super Sub>。
- 不管是协变类型的泛型类 Bar<out T>，还是逆变类型的泛型类 Foo<in T>，当它作为返回值出现时，Kotlin 不会生成通配符。

例如如下 Kotlin 程序。

程序清单：codes\12\12.2\GenericVariant.kt

```
open class Base
class Sub: Base()
class Box<out T>(val value: T)

fun boxSub(value: Sub): Box<Sub> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

上面程序中定义了一个支持声明处协变的 Box 类，接下来程序定义了两个函数，分别使用 Box<Base> 作为参数类型，使用 Box<Sub>作为返回值类型。编译上面程序，可以看到编译器生成的两个函数的签名如下：

```
public static final Box<Sub> boxSub(Sub)
public static final Base unboxBase(Box<? extends Base>)
```

从上面粗体字代码可以看出，由于 Box<out T>是声明处协变的泛型类，因此程序将它作为参数时，编译器自动使用了运行处协变来代替它。这样 Java 程序调用 unboxBase()方法时既可传入 Box<Base>作为参数，也可传入 Box<Sub>作为参数，只要尖括号中的类型是 Base 的子类即可。

除 Kotlin 默认的转换规则之外，Kotlin 还可使用注解控制是否生成通配符。Kotlin 提供了如下两个注解。

- @JvmWildcard：该注解可指定在编译器默认不生成通配符的地方强制生成通配符。
- @JvmSuppressWildcards：该注解可指定在编译器默认生成通配符的地方强制不生成通配符。

如下程序示范了@JvmWildcard 注解的用法。

程序清单：codes\12\12.2\MustWildcard.kt

```
open class Base
class Sub: Base()
class Box<out T>(val value: T)

// 对返回值类型强制生成通配符
fun boxSub(value: Sub): Box<@JvmWildcard Sub> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

上面粗体字代码使用@JvmWildcard 修饰了该函数的返回值类型：Box<Sub>，因此编译器将会为该返回值类型生成通配符。因此，上面两个函数编译之后，生成两个函数的签名如下：

```
public static final Box<? extends Sub> boxSub(Sub)
public static final Base unboxBase(Box<? extends Base>)
```

从上面代码可以看出，此处编译器为返回值类型也生成了通配符（默认是不会生成通配符的）。

如下程序示范了@JvmSuppressWildcards 注解的用法。

程序清单：codes\12\12.2\MustNotWildcard.kt

```
open class Base
class Sub: Base()
class Box<out T>(val value: T)

// 对返回值类型强制不生成通配符
fun boxSub(value: Sub): Box<Sub> = Box(value)
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
```

上面粗体字代码使用@JvmSuppressWildcards 修饰了该函数的形参类型：Box<Base>，因此编译器将不会为函数的形参类型生成通配符。因此，上面两个函数编译之后，生成两个函数的签名如下：

```
public static final Box<Sub> boxSub(Sub)
public static final Base unboxBase(Box<Base>)
```

从上面代码可以看出，此处编译器没有为形参类型生成通配符（默认会生成通配符）。



提示：

@JvmSuppressWildcards 注解不仅可修饰单个泛型形参，还可修饰整个声明（如函数或类），从而阻止编译器为整个类或函数中的声明处型变生成通配符（使用处协变）。

12.3 Kotlin 反射

正如在前面不少代码中所见到的，Kotlin 也提供了反射 API，这些反射 API 可以方便程序在运行时自省程序的结构。Kotlin 把函数和属性当成“一等公民”，并可通过反射直接获取函数、属性的引用。



提示：

使用 Kotlin 的反射 API 需要添加单独的 JAR 文件（kotlin-reflect.jar），这样可以方便程序在不使用反射时减小运行库的大小。

12.3.1 类引用

Kotlin 的类引用使用 KClass 代表，如果要获取已知的 Kotlin 类的 KClass 对象，则可通过如下语法：

```
val c = MyClass::class
```

上面这种语法在前面程序中已经多次见到，此处不再给出示例程序。

需要说明的是，Kotlin 的类引用是 KClass 对象，Java 的类引用是 java.lang.Class 对象，它们二者是不同的，如果需要通过 KClass 获取对应的 java.lang.Class 对象，则可调用 KClass 对象的 java 属性。

如果已有一个 Kotlin 对象，则同样可通过::class 语法来获取该对象的类引用。例如如下

语法：

```
val c = myObj::class
```

12.3.2 从 KClass 获取类信息

获取 KClass 对象之后，即可通过 KClass 提供的大量方法或属性来获取该 KClass 对象所对应类的详细信息，读者应该查阅 API 文档来掌握它们。



提示：

KClass 类的文档可参考 https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-class/index.html。

下面程序示范了通过 KClass 来获取类的详细信息。该程序示范了 KClass 所包含的部分属性和方法，但更详细的用法还是需要读者自行查阅 API 文档。



提示：

下面的反射程序大量使用了 Kotlin 1.1 为反射扩展的新 API，因此必须导入 kotlin.reflect.full.*包。

程序清单：codes\12\12.3\KClassTest.kt

```
import kotlin.reflect.full.*
// 定义注解
annotation class Anno
// 使用 3 个注解修饰该类
@Deprecated("该类已经不推荐使用")
@Anno
@Suppress("UNCHECKED_CAST")
class ClassTest(age: Int){
    var name: String = "Kotlin"
    // 为该类定义一个私有的构造器
    private constructor(): this(20){
    }
    // 定义一个有参数的构造器
    constructor(name: String): this(15){
        println("执行有参数的构造器:${name}")
    }
    // 定义一个无参数的 info 方法
    fun info(){
        println("执行无参数的 info 方法")
    }
    // 定义一个有参数的 info 方法
    fun info(str: String){
        println("执行有参数的 info 方法"
            + ", 其 str 参数值: " + str)
    }
    // 定义一个测试用的嵌套类
    class Inner
}
// 为 ClassTest 定义扩展方法
fun ClassTest.bar() {
    println("扩展的 bar 方法")
}
// 为 ClassTest 定义扩展属性
```

```
val ClassTest.foo: Double
    get() = 2.4
fun main(args: Array<String>) {
    // 下面代码可以获取 ClassTest 对应的 KClass
    val clazz = ClassTest::class
    // 通过 constructors 属性获取 KClass 对象所对应类的全部构造器
    val ctors = clazz.constructors;
    println("ClassTest 的全部构造器如下：")
    ctors.forEach { println(it) }
    println("ClassTest 的主构造器如下：")
    println(clazz.primaryConstructor)

    // 通过 functions 属性获取该 KClass 对象所对应类的全部方法
    var funs = clazz.functions
    println("ClassTest 的全部方法如下：")
    funs.forEach { println(it) }

    /*
    通过 declaredFunctions 属性获取该 KClass 对象
    本身所声明的全部方法（不包括继承的方法） */
    var funs2 = clazz.declaredFunctions
    println("ClassTest 本身声明的全部方法如下：")
    funs2.forEach { println(it) }

    /*
    通过 declaredMemberFunctions 属性获取该 KClass 对象
    本身所声明的全部成员方法（不包括继承的方法） */
    var memberFunctions = clazz.declaredMemberFunctions
    println("ClassTest 本身声明的成员方法如下：")
    memberFunctions.forEach { println(it) }

    /*
    通过 memberExtensionFunctions 属性获取该 KClass 对象
    所代表类的全部扩展方法（不包括继承的方法） */
    var extensionFunctions = clazz.memberExtensionFunctions
    println("ClassTest 本身声明的扩展方法如下：")
    extensionFunctions.forEach { println(it) }

    /*
    通过 declaredMemberProperties 属性获取该 KClass 对象
    本身所声明的全部成员属性（不包括继承的属性） */
    var memberProperties = clazz.declaredMemberProperties
    println("ClassTest 本身声明的成员属性如下：")
    memberProperties.forEach { println(it) }

    /*
    通过 memberExtensionProperties 属性获取该 KClass 对象
    所代表类的全部扩展属性（不包括继承的属性） */
    var extensionProperties = clazz.memberExtensionProperties
    println("ClassTest 本身声明的扩展属性如下：")
    extensionProperties.forEach { println(it) }

    // 通过 annotations 属性获取该 KClass 对象所对应类的全部注解
    val anns = clazz.annotations;
    println("ClassTest 的全部注解如下：")
    anns.forEach { println(it) }
    println("该 KClass 元素上的@Anno 注解为："
```

```
        + clazz.findAnnotation<Anno>())
    // 通过 nestedClasses 属性获取该 KClass 对象所对应类的全部嵌套类（包括内部类和嵌套类）
    val inners = clazz.nestedClasses
    System.out.println("ClassTest 的全部内部类如下：")
    inners.forEach { println(it) }
    // 通过 supertypes 属性获取该类的所有父类型（包括父类和父接口）
    println("ClassTest 的父类型为：" + clazz.supertypes)
}
```

上面程序无须过多解释，注释已经写得非常清楚，程序获取了 ClassTest 类对应的 KClass 对象后，通过调用该 KClass 对象的不同方法来得到其详细信息。运行该程序，将会看到如图 12.1 所示的运行结果。

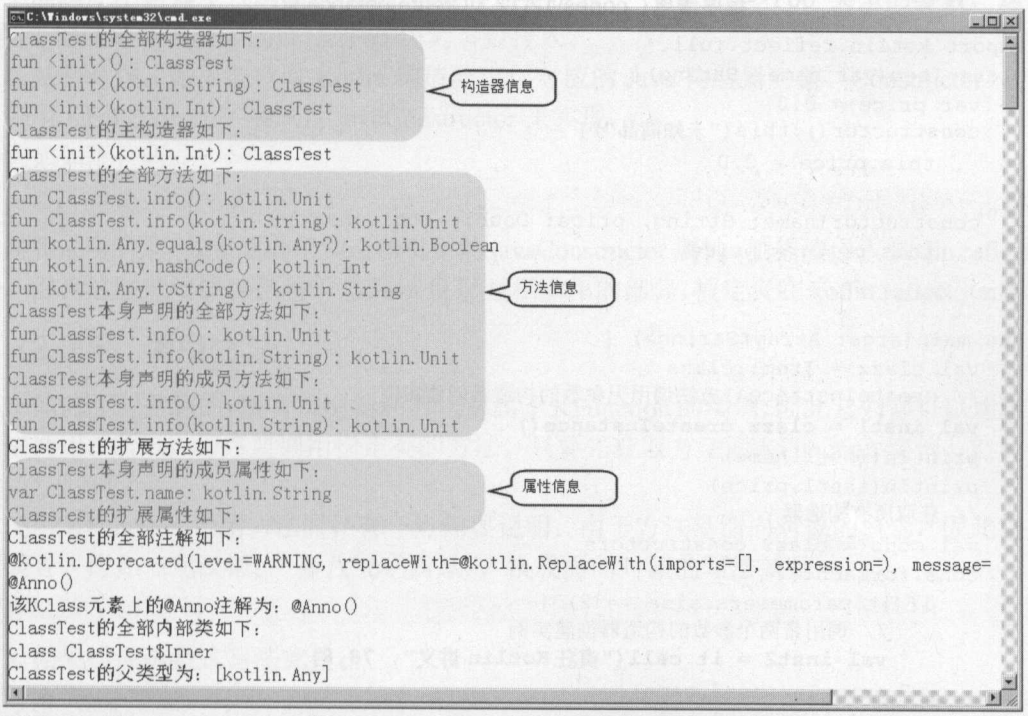


图 12.1　使用 KClass 对象查看该类的内部信息

从图 12.1 所示的运行结果来看，KClass 提供的功能非常丰富，可以获取该类中包含的构造器、方法、属性、内部类、注解等信息。

值得指出的是，虽然定义 ClassTest 类时使用了@Suppress 注解，但程序运行时无法分析出该类中包含的该注解，这是因为@Suppress 使用了@Retention(SOURCE)修饰，这表明@Suppress 只能保存在源代码级别上，而通过 ClassTest.class 获取的是该类的运行时 KClass 对象，所以程序无法访问到@Suppress 注解。



提示：对于只能在源代码上保留的注解，使用运行时获得的 KClass 对象无法访问到该注解对象。

通过 KClass 对象可以得到大量的 KFunction、KProperty（它们都是 KCallable 的子类）等对象，这些对象分别代表该类所包括的方法（包括构造器）和属性等，程序还可以通过这些对象来执行实际的功能，例如调用方法、创建实例等。

12.3.3 创建对象

获取 KClass 对象之后，调用该对象的 createInstance()方法即可创建该类的实例，该方法总是调用 KClass 所代表类的无参数的构造器来创建实例。

如果需要调用有参数的构造器来创建实例，则可通过 KClass 的 constructors 属性来获取所有构造器，该属性返回 Collection<Function>集合对象，这意味着构造器的本质依然是一个函数。

获取 KClass 的所有构造器之后，接下来程序可根据需要调用指定的构造器来创建实例。下面程序示范了如何通过 KClass 创建实例。

程序清单：codes\12\12.3\CreateInstance.kt

```
import kotlin.reflect.full.*
class Item(var name: String){
    var price = 0.0
    constructor():this("未知商品"){
        this.price = 0.0
    }
    constructor(name: String, price: Double):this(name){
        this.price = price
    }
}
fun main(args: Array<String>) {
    val clazz = Item::class
    // createInstance()方法调用无参数的构造器创建实例
    val inst1 = clazz.createInstance()
    println(inst1.name)
    println(inst1.price)
    // 获取所有构造器
    val cons = clazz.constructors
    cons.forEach {
        if(it.parameters.size == 2) {
            // 调用带两个参数的构造器创建实例
            val inst2 = it.call("疯狂 Kotlin 讲义", 78.8)
            println(inst2.name)
            println(inst2.price)
        }
    }
}
```

上面程序中第一行粗体字代码调用 KClass 的无参数的构造器创建实例，第二行粗体字代码调用带两个参数的构造器来创建实例。编译、运行该程序，可看到如下结果：

```
未知商品
0.0
疯狂 Kotlin 讲义
78.8
```

12.3.4 构造器引用

正如前文所介绍的，构造器的本质是一个函数，即一个返回值为当前类实例的函数。因此程序可将构造器引用当成函数使用。

此外，Kotlin 允许通过使用“::”操作符并添加类名来引用该类的主构造器。例如如下程序。

程序清单：codes\12\12.3\ConstructorRef.kt

```
class Foo(var name: String = "未知")
// test 函数的参数是(String)->Foo 类型（这就是 Foo 带 String 参数的构造器的类型）
fun test(factory: (String) -> Foo) {
    val x: Foo = factory("疯狂 Kotlin 讲义")
    println(x.name)
}
fun main(args: Array<String>) {
    // 通过::Foo 引用 Foo 类的主构造器
    test(::Foo)
}
```

上面程序中粗体字代码调用 test()函数时需要传入一个(String)->Foo 类型的参数，这就是 Foo 类主构造器的类型，因此将::Foo 作为参数传入。

在某些时候，如果要获取 Kotlin 构造器引用对应的 Java 构造器对象（Constructor），则可通过调用 KFunction 的扩展属性 javaConstructor 来实现。

例如如下代码：

```
::Foo.javaConstructor
```

需要说明的是，如果要调用构造器引用的 javaConstructor 属性，则需要导入 kotlin.reflect.jvm 包——因为这些扩展属性都属于与 Java 反射互相操作的部分，被定义在 kotlin.reflect.jvm 包下。

12.3.5 调用方法

正如前面所见到的，所有构造器和方法都属于 KFunction 的实例，因此它们都可以通过 call()方法来调用。所以，程序要调用指定类的方法，只要先获取方法的 KFunction 实例，然后调用 call()方法即可。

使用 KFunction 调用方法时，有一点需要说明：由于方法是面向对象的概念，因此它有一个主调者。比如一句汉语：“猪八戒吃西瓜”，换成面向对象的写法就是：

```
猪八戒.吃(西瓜)
```

但如果换成函数式写法就是：

```
吃(猪八戒, 西瓜)
```

对比这两种写法可以看出，面向对象的方法如果带 *N* 个参数，那么转换成函数式调用时就会变成 *N*+1 个参数。

如下程序示范了调用指定函数。

程序清单：codes\12\12.3\CallFunction.kt

```
import kotlin.reflect.full.*
class Foo{
    fun test(msg: String){
        println("执行带 String 参数的 test 方法：${msg}")
    }
    fun test(msg: String, price: Double){
        println("执行带 String, Double 参数的 test 方法：${msg}, ${price}")
    }
}
fun main(args: Array<String>) {
    val clazz = Foo::class
    // 创建 Foo 类的实例
    val ins = clazz.createInstance()
```

```
    // 获取 clazz 所代表类直接定义的全部函数
    val funs = clazz.declaredFunctions
    for (f in funs) {
        // 如果函数具有 3 个参数（对应带 2 个参数的方法）
        if (f.parameters.size == 3) {
            // 调用带 3 个参数的函数
            f.call(ins, "Kotlin", 78.8)
        }
        // 如果函数具有 2 个参数（对应带 1 个参数的方法）
        if (f.parameters.size == 2) {
            // 调用带 2 个参数的函数
            f.call(ins, "Kotlin")
        }
    }
}
```

从上面程序可以看出，执行带 3 个参数的函数实际上是调用带 2 个参数的方法，执行带 2 个参数的函数实际上是调用带 1 个参数的方法，方法的调用者将作为函数的第一个参数被传入。

12.3.6 函数引用

Kotlin 的函数也是一等公民，函数也有其自身的类型。Kotlin 程序可以获取函数的引用，把函数当成参数传入另一个函数中。

Kotlin 也通过“::”符号加函数名的形式来获取特定函数的引用。当存在多个重载函数时，Kotlin 可通过上下文推断出实际引用的是哪个函数；如果 Kotlin 无法通过上下文准确推断出引用哪个函数，编译器就会报错。

例如如下程序。

程序清单：codes\12\12.3\FunctionRef.kt

```
// 定义两个重载的函数
fun isSmall(i: Int) = i < 5
fun isSmall(s: String) = s.length < 5
fun main(args: Array<String>) {
    val list = listOf(20 , 30, 100, 4, -3, 2, -12)
    /*
    由于 filter()函数需要(Int)->Boolean 类型的参数，
    故此处::isSmall 引用第一个函数
    */
    val resultList = list.filter(::isSmall)
    println(resultList) // 输出[4, -3, 2, -12]
    val strlist = listOf("Java" , "Kotlin", "Swift", "Go", "Erlang")
    /*
    由于 filter()函数需要(String)->Boolean 类型的参数，
    故此处::isSmall 引用第二个函数
    */
    val resultStrList = strlist.filter(::isSmall)
    println(resultStrList) // 输出[Java, Go]
    // 无法推断出::isSmall 到底引用哪个函数，报错
//    val f = ::isSmall
    // 可以推断出::isSmall 到底引用哪个函数，正确
    var f: (String) -> Boolean = ::isSmall
    println(f("Lua"))
}
```

正如从上面两行粗体字代码所看到的，当 List 集合是 List<String>实例时，它的 filter()方

法需要的参数是(String)->Boolean 类型，因此 Kotlin 可以准确地从重载函数中引用到符合要求的函数。

此外需要说明的是，如果需要引用类的成员方法或扩展方法，那么需要进行限定。例如 String::toCharArray 才能表明引用 String 的 toCharArray()方法，单纯地使用::toCharArray()不行。

String::toCharArray 函数引用的类型也不是简单的()->CharArray 类型，而是 String.() -> CharArray 类型。

有些时候，程序需要实现某个功能较强的函数，如果此时系统已经包含了多个细粒度的函数，那么可以将这些细粒度的函数组合起来实现功能较强的函数。比如业务需要程序获取数的平方根，但该函数要做一些额外处理——如果该数是正数，则直接获取平方根；如果该数是负数，则获取该数的绝对值的平方根。

下面程序示范了如何利用函数引用来组合两个函数。

程序清单：codes\12\12.3\FunctionComposition.kt

```
fun abs(d: Double): Double = if (d < 0) -d else d
fun sqrt(d: Double): Double = java.lang.Math.sqrt(d)
// 定义一个 comp()函数，该函数用于将两个函数组合起来
fun comp(fun1: (Double) -> Double, fun2: (Double) -> Double) :
    (Double) -> Double{
    return { x -> fun2(fun1(x)) }
}
fun main(args: Array<String>) {
    println(abs(-3.2))
    // 将::abs 和::sqrt 组合起来
    val f = comp(::abs, ::sqrt)
    println(f(-25.0))
}
```

上面程序中粗体字代码用 comp()函数将::abs 和::sqrt 两个函数组合在一起，这样就会得到一个新的函数：f，接下来程序可通过 f()函数同时完成两个函数的功能。

在某些时候，如果要获取 Kotlin 函数引用对应的 Java 方法对象（Method），则可通过调用 KFunction 的扩展属性 javaMethod 来实现。

例如如下代码：

```
::abs.javaMethod
```

需要说明的是，如果要调用函数引用的 javaMethod 属性，则需要导入 kotlin.reflect.jvm 包——因为这些扩展属性都属于与 Java 反射互相操作的部分，被定义在 kotlin.reflect.jvm 包下。

12.3.7 访问属性值

获取 KClass 对象之后，也可通过 KClass 对象来获取该类所包含的属性。Kotlin 为属性提供了众多的 API。

- KProperty：代表通用的属性。它是 KCallable 的子接口。
- KMutableProperty：代表通用的读写属性。它是 KProperty 的子接口。
- KProperty0：代表无需调用者的属性（静态属性）。它是 KProperty 的子接口。
- KMutableProperty0：代表无需调用者的读写属性（静态读写属性）。它是 KProperty0 的子接口。
- KProperty1：代表需要 1 个调用者的属性（成员属性）。它是 KProperty 的子接口。
- KMutableProperty1：代表需要 1 个调用者的读写属性(成员读写属性)。它是 KProperty1

的子接口。

- KProperty2：代表需要2个调用者的属性（扩展属性）。它是KProperty的子接口。
- KMutableProperty2：代表需要2个调用者的读写属性（扩展读写属性）。它是KProperty2的子接口。

它们之间的继承关系如图12.2所示。

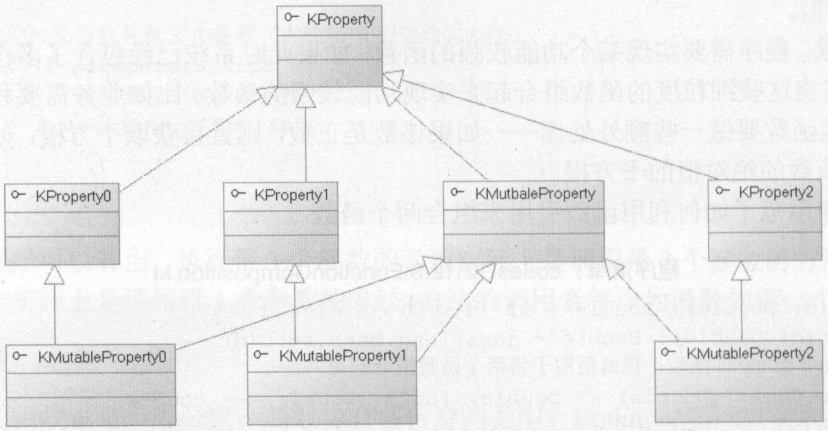


图12.2 KProperty各子接口之间的继承关系

程序获取代表属性的KProperty对象之后，可调用get()方法来获取属性的值；如果程序要设置属性的值，则需要获取代表属性的KMutableProperty对象。

如下程序示范了如何通过反射来设置和获取属性的值。

程序清单：codes\12\12.3\CallProperty.kt

```
import kotlin.reflect.*
import kotlin.reflect.full.*
class Item{
    var name: String = "疯狂Kotlin讲义"
    val price: Double = 76.6
}
fun main(args: Array<String>) {
    val clazz = Item::class
    val ins = clazz.createInstance()
    val props = clazz.declaredMemberProperties
    props.forEach {
        when (it.name) {
            "name" -> {
                @Suppress("UNCHECKED_CAST")
                // 将属性转换为读写属性
                val mp = it as KMutableProperty1<Item, Any>
                // 修改属性值
                mp.set(ins, "疯狂Java讲义")
                println(it.get(ins))
            }
            "price" -> {
                // 只读属性，只能通过get()方法读取属性值
                println(it.get(ins))
            }
        }
    }
}
```

正如从上面粗体字代码所看到的，当程序要设置 name 属性的属性值时，由于 name 属性是一个成员读写属性，因此程序将该属性转型为 KMutableProperty1 对象，转型之后程序可调用 set()方法来设置该属性的值，如上面程序中前两行粗体字代码所示；如果程序只需获取该属性的值，则调用 KProperty1 的 get()方法即可，如上面程序中第三行粗体字代码所示。

12.3.8 属性引用

Kotlin 同样提供了“::”符号加属性名的形式来获取属性引用，获取属性引用也属于前面介绍的 KProperty 及其子接口的实例。

获取 Kotlin 只读属性的引用之后，程序可调用 get()方法来获取属性的值；获取 Kotlin 读写属性的引用之后，程序可调用 set()方法来修改属性的值，也可调用 get()方法来获取属性的值。

如下程序示范了通过属性引用来操作属性。

程序清单：codes\12\12.3\PropertyRef.kt

```
import kotlin.reflect.*
import kotlin.reflect.full.*
class Item{
    var name: String = "疯狂 Kotlin 讲义"
    val price: Double = 76.6
}
var foo = "测试属性"
fun main(args: Array<String>) {
    // 获取 foo 属性，属于 KMutableProperty0 的实例
    val topProp = ::foo
    topProp.set("修改后的属性")
    println(topProp.get())

    val im = Item()
    // 获取 Item 的 name 属性，属于 KMutableProperty1 的实例
    val mp = Item::name
    mp.set(im, "疯狂 Java 讲义")
    println(mp.get(im))

    // 获取 Item 的 price 属性，属于 KProperty1 的实例
    val prop = Item::price
    println(prop.get(im))
}
```

上面程序中第一行粗体字代码获取顶级读写属性（静态读写属性），程序直接用“::”加属性名的形式即可。该属性引用是 KMutableProperty0 的实例，因此该属性既可通过 set()方法改变属性的值，也可通过 get()方法获取属性的值。

第二行粗体代码用于获取 Item 类的 name 读写属性，程序需要用类名::属性名的形式来获取指定类的读写属性。该属性是 KMutableProperty1 的实例，因此程序也可通过 set()、get()方法来修改、获取属性的值。

第三行粗体代码用于获取 Item 类的 price 只读属性，程序需要用类名::属性名的形式来获取指定类的只读属性，程序只能通过 get()方法来获取属性的值。

与前面介绍的构造器、函数相似，Kotlin 在 kotlin.reflect.jvm 包下也提供了 Kotlin 属性与

Java 反射互操作的扩展属性。由于 Kotlin 属性会对应于 Java 的 3 种成员，因此 KProperty 包含如下 3 个扩展属性。

- javaField：获取该属性的幕后字段（如果该属性有幕后字段的话）。该属性返回 java.lang.reflect.Field 对象。
- javaGetter：获取该属性的 getter 方法。该属性返回 java.lang.reflect.Method 对象。
- javaSetter：获取该属性的 setter 方法（如果该属性是读写属性的话）。该属性返回 java.lang.reflect. Method 对象。

一旦获取了 Kotlin 属性的幕后字段的 Field 对象、getter 和 setter 方法的 Method 对象之后，剩下的就是 Java 反射的事了。

如下程序示范了通过 Kotlin 属性来获取 Java 反射 API。

程序清单：codes\12\12.3\PropertyRef2java.kt

```
import kotlin.reflect.*
import kotlin.reflect.full.*
import kotlin.reflect.jvm.*
class Item{
    var name: String = "疯狂 Kotlin 讲义"
    val price: Double = 76.6
}
var foo = "测试属性"
fun main(args: Array<String>) {
    // 获取 foo 属性，属于 KMutableProperty0 的实例
    val topProp = ::foo
    println(topProp.javaField) // 获取幕后字段
    println(topProp.javaGetter) // 获取 getter 方法
    println(topProp.javaSetter)    // 获取 setter 方法

    // 获取 Item 的 name 属性，属于 KMutableProperty1 的实例
    val mp = Item::name
    println(mp.javaField) // 获取幕后字段
    println(mp.javaGetter) // 获取 getter 方法
    println(mp.javaSetter)    // 获取 setter 方法

    // 获取 Item 的 price 属性，属于 KProperty1 的实例
    val prop = Item::price
    println(prop.javaField) // 获取幕后字段
    println(prop.javaGetter) // 获取 getter 方法
}
```

上面程序中第一行粗体字代码用于获取 Kotlin 属性的幕后字段；第二行粗体字代码用于获取 Kotlin 属性 getter 方法；第三行粗体字代码用于获取 Kotlin 属性的 setter 方法。它们返回的都是 java.lang.reflect 包下的反射 API。

编译、运行该程序，可看到如下输出：

```
private static java.lang.String PropertyRef2JavaKt.foo
public static final java.lang.String PropertyRef2JavaKt.getFoo()
public static final void PropertyRef2JavaKt.setFoo(java.lang.String)
private java.lang.String Item.name
public final java.lang.String Item.getName()
public final void Item.setName(java.lang.String)
```

```
private final double Item.price
public final double Item.getPrice()
```

12.3.9 绑定的方法与属性引用

前面介绍的都是通过 KClass（类本身）来获取方法或属性的引用的，当函数或属性不在任何类中定义时，程序直接使用“::”加函数名（或属性名）的形式来获取函数或属性的引用，这些函数或属性都没有绑定任何对象，因此调用函数或属性时第一个参数必须传入调用者。

从 Kotlin 1.1 开始，Kotlin 支持一种“绑定的方法或属性引用”，这种方法或属性引用不是通过类获取的，而是通过对象获取的，这意味着该方法或属性已经绑定了调用者，因此程序执行这种方法或属性时无须传入调用者。

如下程序示范了绑定的方法或属性引用。

程序清单：codes\12\12.3\BoundMember.kt

```
fun main(args: Array<String>) {
    val str = "Kotlin"
    // 获取对象绑定的方法
    val f: (CharSequence, Boolean)->Boolean = str::endsWith
    // 调用绑定的方法时无须传入调用者
    println(f("lin" , true))
    // 获取对象绑定的属性
    val prop = str::length
    // 调用绑定的属性时无须传入调用者
    println(prop.get())

    var list = listOf("Kotlin", "Java", "Go", "Erlang")
    // 获取对象绑定的方法
    val fn = list::subList
    // 调用绑定的方法时无须传入调用者
    println(fn(1, 3)) // 输出["Java", "Go"]
    // 获取对象绑定的属性
    val prp = list::indices
    // 调用绑定的属性时无须传入调用者
    println(prp.get()) // 输出 0..3
}
```

上面程序先定义了一个简单的字符串，其中第一行粗体字代码使用对象来获取绑定的方法，这样当程序调用方法时只要传入该方法的两个参数即可，无须传入调用者；第二行粗体字代码使用对象来获取绑定的属性，这样程序直接使用 get()方法即可获取绑定的属性值，无须传入调用者。

上面程序中第二段代码只是示范了 List 集合的绑定方法和属性引用，与前一段代码基本相似。

12.4 本章小结

本章主要介绍了 Kotlin 与 Java 的相互调用，以及反射相关内容。由于 Kotlin 在设计之初就已考虑到 Kotlin 与 Java 的相互调用，因此 Kotlin 与 Java 相互调用是非常平滑的，只是读者在使用两种语言相互调用时略微注意一下它们的差异点即可。常见的差异点有属性、类方法

（Kotlin 没有 static 关键字）、泛型型变（Kotlin 支持声明处型变）、数组型变（Kotlin 的数组不型变，Java 的数组型变）、checked 异常等，因此本章详细介绍了这些差异点在相互调用中可能会出现的问题。

此外，本章的另外一个重点内容是 Kotlin 的反射。Kotlin 本身提供了一套反射 API：KClass、KCallable、KFunction、KProperty 及其各种子类，通过这套反射 API 可以让程序在运行时自省程序的内容。而且这套 API 与 Java 的反射 API 也可以相互转换，因此读者需要掌握 Kotlin 反射 API 的用法，以及 Kotlin 反射 API 和 Java 反射 API 之间的转换关系。

疯狂Java学习路线图（第二版）

说明：

1. 没有背景色覆盖的区域稍有难度，请谨慎尝试。
2. 路线图上背景色与对应教材的封面颜色相同。
3. 已发现不少培训机构抄袭、修改该学习路线图，务请各培训机构保留对路线图的名称、引用说明。



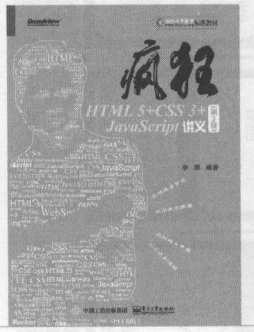
疯狂Java体系

疯狂源自梦想 技术成就辉煌



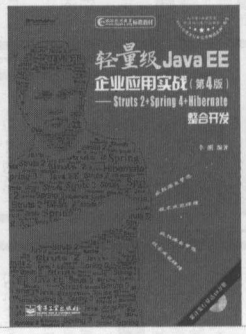
疯狂Java程序员的基本修养

作　　者：李刚
定　　价：59.00元
出版时间：2013-01
书　　号：978-7-121-19232-6



疯狂HTML 5＋CSS 3＋JavaScript 讲义（第2版）

作　　者：李刚
定　　价：89.00元
出版时间：2017-05
书　　号：978-7-121-31405-6



轻量级Java EE企业应用实战（第4版）——Struts 2＋Spring4＋Hibernate整合开发

作　　者：李刚
定　　价：108.00元（含光盘1张）
出版时间：2014-10
书　　号：978-7-121-24253-3



经典Java EE企业应用实战——基于WebLogic/JBoss的JSF+EJB 3+JPA整合开发

作　　者：李刚
定　　价：79.00元（含光盘1张）
出版时间：2010-08
书　　号：978-7-121-11534-9



疯狂Java讲义（第4版）

作　　者：李刚
定　　价：109.00元（含光盘1张）
出版时间：2018-01
书　　号：978-7-121-33108-4



疯狂前端开发讲义——jQuery+AngularJS+Bootstrap前端开发实战

作　　者：李刚
定　　价：79.00元
出版时间：2017-10
书　　号：978-7-121-32680-6



疯狂XML讲义（第2版）

作　　者：李刚
定　　价：69.00元（含光盘1张）
出版时间：2011-08
书　　号：978-7-121-14049-5



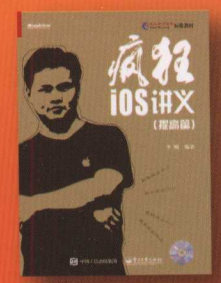
疯狂Android讲义（第3版）

作　　者：李刚
定　　价：108.00元（含光盘1张
出版时间：2015-06
书　　号：978-7-121-25958-6

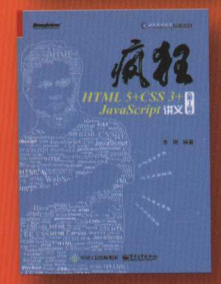
新浪微博：weibo.com/crazyjavabooks @疯狂Java体系图书

Li Gang's Series

李刚 作品系列









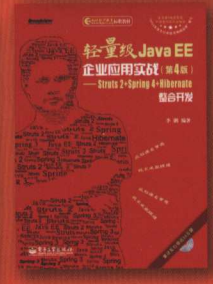












Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

疯狂源自梦想　技术成就辉煌

1. 逻辑结构更合理

本书在内容体系上将函数式编程和面向对象编程独立开来，先介绍函数式编程部分，再介绍面向对象编程，更符合Kotlin语言本身的知识体系。而不像某些资料一会儿函数，一会儿面向对象，搅得读者晕头转向。

在介绍知识时，本书会先详细讲解各种知识点的理论，然后再通过示例演示Kotlin各理论的用法，将知识点融合在示例中，符合读者的认知、学习规律。

2. 讲解深入本质

Kotlin是JVM语言，所以其很多东西其实是受到Java的影响的。书中会对一些看似奇怪的语法从字节码文件层次进行剖析，让读者更好地理解Kotlin与Java的对应关系，从而以较低成本快速掌握Kotlin。

3. 知识内容更全面

本书内容超过Kotlin官方文档本身所覆盖的知识，比如介绍反射的章节就超过了Kotlin官方文档内容。本书反射部分不仅更详细地介绍如何获得类、函数、属性的引用，而且真正从API级别介绍KClass、KCallable、KFunction、KProperty，KProperty0、KProperty1、KProperty2的用法，以及它们的内在关联，并实实在在地教读者掌握如何用Kotlin反射动态创建对象、动态调用方法。

阅读此书如有技术问题或欲获取本书配套资源，可登录http://www.crazyit.org
亦可访问http://www.broadview.com.cn/33459 下载配套资源

上架建议：编程语言>Kotlin

ISBN 978-7-121-33459-7

定价：69.00元

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：张月萍
责任编辑：葛　娜
封面设计：李　玲